Operating Systems Project: Topic 2

Kernel-Level Thread Implementation and Analysis

Liangsen Wang

224040364@link.cuhk.edu.cn

2026.1.12

Outline

1. Foundations: Core Concepts
2. Theory: Evolution of Linux Threading
3. Mechanism: The `clone()` System Call
4. Kernel API & Lifecycle Management
5. Project Topic 2 Requirements

Outline

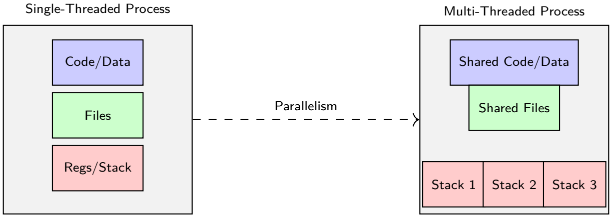
Foundations: Process vs. Thread

Process (Resource Container)

- An instance of a running program.
- **Isolated Resources:** Owns Memory (Page Tables), File Descriptors, Signals.
- **Heavyweight:** Creation requires duplicating the entire address space (COW helps, but page tables are still copied).

Thread (Execution Unit)

- "Lightweight Process" (LWP).
- **Shared Resources:** Code, Global Data, Heap, Open Files.
- **Private Resources:** Program Counter (PC), Register Set, **Stack**.



Foundations: User-Level Threads (ULT)

Definition: Threads managed entirely by a user-space library (e.g., Green Threads, GNU Pth). The Kernel knows nothing about them.

Mechanism

- Thread Table is stored in process memory.
- **Context Switch:** Just saving registers to user stack. No Mode Switch (User $\leftrightarrow$ Kernel).
- **Speed:** Extremely fast ($10\times$ faster than KLT).

The Fatal Flaw

Blocking System Calls: If one ULT executes a blocking syscall (e.g., `read()`), the **Kernel blocks the entire process**.

Result: All other threads stop, even if they are ready to run.

Foundations: Kernel-Level Threads (KLT)

Definition: Threads managed directly by the OS Kernel. (This is what we use in modern Linux).

Mechanism

- Kernel maintains a Thread Table (TCB) for every thread.
- **Scheduling:** Kernel schedules threads, not processes.
- **Parallelism:** Can run on different CPU cores simultaneously.

Trade-offs

- **Pros:** If one thread blocks, others continue running. True Multi-core utilization.
- **Cons:** Context switch requires a **Mode Switch** (User $\rightarrow$ Kernel $\rightarrow$ User), which is expensive (cache pollution, TLB implications).

Outline

The Big Picture: Threading Models

Before diving into Linux, we must understand the three mapping models:

1. **Many-to-One (M:1)**
   - User-Level Threads.
   - Kernel sees 1 process.
   - *Example: Old Java Green Threads.*
   - Block one = Block all.

2. **One-to-One (1:1)**
   - **Linux Model (NPTL).**
   - 1 User Thread = 1 Kernel Entity.
   - *Example: Pthreads on Linux.*
   - True Parallelism.

3. **Many-to-Many (M:N)**
   - Hybrid approach.
   - Complex scheduler in user space.
   - *Example: Go Goroutines, Erlang.*
   - High concurrency, low overhead.

History: From "Hack" to Standard

The Old Days: LinuxThreads (Pre-2.6)

- Used `clone()` but logic was flawed.
- Signal Handling Issue: Signals were sent to specific threads, not the process.
- PID Issue: Each thread had a different PID (broke POSIX compliance).

The Modern Standard: NPTL (Native POSIX Thread Library)

- Introduced in kernel 2.6.
- 1:1 Model: Kernel manages scheduling directly.
- Solved the PID/Signal issues using `Thread Group ID (TGID)`.
- Heavy optimization for context switching.

The Core Concept: PID vs. TGID

Crucial for your understanding of kernel internals:

- **User Space View (`getpid()`):**
  - All threads in a process share the **Same PID**.
- **Kernel View (`task_struct`):**
  - Each thread is a "Task".
  - Each task has a unique ID: **`pid`** (userspace calls this TID).
  - All threads share a **`tgid`** (Thread Group ID).

```
struct task_struct {
    pid_t pid;  // Unique per thread
    pid_t tgid; // Shared by group

    struct task_struct *group_leader
    ;
    struct list_head thread_group;
};

```

Logic: If `pid == tgid`, this task is the Main Thread.

Outline

Creating Processes vs. Threads

In Linux, there is no "Create Thread" system call. There is only "Create Task".

System Call	Semantics	Sharing
`fork()`	Copy-on-Write (COW)	Shares nothing (mostly). Copies page tables.
`vfork()`	Block Parent	Shares memory (legacy, dangerous).
`clone()`	**Flexible**	**Selective sharing** via flags.

Project Insight: When you implement "Kernel Threads", you are essentially creating a task that shares **Kernel Memory** but has its own stack.

Deep Dive: clone() Flags

Understanding these flags is required to understand kernel/fork.c.

```c
// Essential Flags for Threads (pthread_create uses these)
#define CLONE_VM        0x00000100  // Share Memory Descriptor (mm_struct)
#define CLONE_FS        0x00000200  // Share Filesystem info (cwd, root)
#define CLONE_FILES     0x00000400  // Share File Descriptor Table (fd)
#define CLONE_SIGHAND   0x00000800  // Share Signal Handlers
#define CLONE_THREAD    0x00010000  // Put in same thread group (Same TGID)

```

Question for Class

If I call clone(CLONE_VM | CLONE_FILES) but **NOT** CLONE_THREAD, what do I get?

Deep Dive: clone() Flags

Understanding these flags is required to understand kernel/fork.c.

```
// Essential Flags for Threads (pthread_create uses these)
#define CLONE_VM        0x00000100  // Share Memory Descriptor (mm_struct)
#define CLONE_FS        0x00000200  // Share Filesystem info (cwd, root)
#define CLONE_FILES     0x00000400  // Share File Descriptor Table (fd)
#define CLONE_SIGHAND   0x00000800  // Share Signal Handlers
#define CLONE_THREAD    0x00010000  // Put in same thread group (Same TGID)

```

Question for Class

If I call clone(CLONE_VM | CLONE_FILES) but **NOT** CLONE_THREAD, what do I get? **Answer:** A classic "Lightweight Process" (LWP) that shares memory but has a different PID in userspace (like old LinuxThreads).

Outline

API: Creating Kernel Threads

Unlike user threads, Kernel Threads (kthreads) have no User Space memory (mm is NULL).
Header: `<linux/kthread.h>`

1. The Easy Way: kthread_run

```c
struct task_struct *ts;
// Creates thread AND wakes it up immediately
ts = kthread_run(thread_fn, data, "worker_%d", id);

```

2. The Manual Way: kthread_create

```c
ts = kthread_create(thread_fn, data, "worker");
if (!IS_ERR(ts)) {
    // You can bind CPU affinity here before starting!
    kthread_bind(ts, cpu_id);
    wake_up_process(ts);
}

```

API: Stopping Kernel Threads

Warning: Killing kernel threads forcefully is bad. They must exit voluntarily.

Worker Logic (Thread Function):

```c
int thread_fn(void *data) {
    while (!kthread_should_stop()) {
        // Do work...

        // IMPORTANT: Yield CPU
        schedule_timeout_interruptible(HZ);
    }
    return 0;
}
```

Controller Logic (Module Exit):

```c
// Request the thread to stop
int ret = kthread_stop(ts);
// This function blocks until thread exits!
```

API: CPU Affinity (Project Requirement)

One of your tasks is to monitor/set CPU Affinity.

Setting Affinity (Binding)

```c
// Bind current thread to CPU 0
kthread_bind(current, 0);

```

Checking Affinity (Statistics)

Inside `task_struct`, look for:

```c
struct cpumask cpus_mask; // Allowed CPUs
int cpu;                  // Current CPU

```

Note: You might need to export this via /proc or printk to verify your scheduler is working.

Outline

Basic Requirements: Implementation Guide

Goal: Create a Kernel Module that spawns threads and measures them.

1. **Module Init:**
   - Create N kernel threads using `kthread_create`.
   - Bind them to specific Cores (e.g., Thread 0 $\rightarrow$ CPU 0).
2. **The Payload:**
   - The threads should do something measurable (e.g., increment a shared atomic counter, or just sleep/wake).
3. **Measurement (The "Analysis" part):**
   - **Context Switches (CSW):** Read `current->nvcsw` (Voluntary) and `current->nivcsw` (Involuntary).
   - Log these values to `dmesg` or a custom `/proc` file before exit.

Requirement: KLT vs ULT Comparison

You must compare your Kernel Module threads against a User-Space Pthread program.

Experiment Setup:

- **Scenario A:** 4 Threads on 4 Cores incrementing a shared atomic variable.
- **Scenario B:** High-frequency yielding (`yield()`).

What to measure?

- **System Time (sys):** High for KLT (syscall overhead if communicating).
- **User Time (user):** Zero for KLT!
- **Throughput:** Operations per second.

Advanced Option: User-Space Coroutines

Challenge: Implement "Green Threads" without Kernel support.
Mechanism: Stack Switching

- You need to allocate a memory block (heap) to act as a stack.
- Use `setjmp` / `longjmp` (C Library) OR `swapcontext` (ucontext.h).

```c
#include <ucontext.h>
ucontext_t main_ctx, thread_ctx;
char stack[16384];

void thread_func() { ... }

// Setup
getcontext(&thread_ctx);
thread_ctx.uc_stack.ss_sp = stack;
thread_ctx.uc_stack.ss_size = sizeof(stack);
makecontext(&thread_ctx, thread_func, 0);

// Switch
swapcontext(&main_ctx, &thread_ctx);

```

Advanced Option: Hybrid Interaction

Goal: Communicate between User Space and Kernel Thread.
Ideas:

- **Netlink Socket:** Asynchronous, message-based. Best for event notification.
- **Debugfs / Sysfs:** Use a file to pass commands.
- **Custom System Call:** (Hardcore) Add a syscall that wakes up a specific wait_queue in the kernel.

Use Case Example: A user program captures network packets (raw socket) and passes them to a high-priority Kernel Thread for processing without copying data (Zero-Copy using shared memory).

Evaluation & Grading (Topic 2 Specific)

Component	Detailed Focus
Implementation (25%)	Correct usage of `kthread_run/stop`. Accurate statistics (CSW, CPU Affinity). Clean cleanup (no zombies).
Advanced Options (10%)	Completeness of User-Space Coroutines (Stack switching) OR Hybrid Communication mechanism OR other innovation.
Stability (5%)	**Crucial:** No Kernel Panics during the live demo. System must remain responsive.
Presentation (20%)	**Demo (10%):** Successful live run. **Structure (5%):** Clarity of technical explanation. **Q&A (5%):** Ability to answer kernel-level questions .
Final Report (20%)	**Guidelines (5%)** **Depth (10%)** **Analysis (5%)**

Resources & Next Steps

Recommended Reading:

- *Linux Kernel Development* (Robert Love) - Chapter 3 (Process Management).
- `man clone`, `man setjmp`.
- Source: `kernel/kthread.c`, `kernel/fork.c`.

Immediate Action Items:

1. Compile a "Hello World" kernel module.
2. Add `kthread_run` to it.
3. Try to unload the module (`rmmod`). If it hangs, you forgot `kthread_should_stop()`!