a tale of needle and `pthread_t`

a novice's weak attempt to explain how processes are woven

2022-10-09 · 17 min · 3963 words

https://tramasys.mov/posts/kernel-threads/

> [!] **updated · august 2026**
>
> I revisited this post against current linux 7.1, glibc 2.44, Rust, libstdc++, and Tokio sources. The boot path, NPTL clone3 path, task model, and x86 context-switch details now follow the current code.

pulling the first thread

The word "thread" is doing too many jobs before we have written any code. There are POSIX threads, linux tasks, thread groups, lightweight processes, kernel threads, hardware threads, and the thread of an argument, which I am already in danger of losing.

I started with a POSIX thread and printed every identifier I could find:

```c
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <sys/syscall.h>
#include <unistd.h>

static void identify(const char *name)
{
    printf("%s: pid=%d tid=%ld pthread=%#lx\n",
           name,
           getpid(),
           syscall(SYS_gettid),
           (unsigned long)pthread_self());
}

static void *worker(void *unused)
{
    identify("worker");
    return NULL;
}

int main(void)
{
    pthread_t thread;

    identify("main");
    pthread_create(&thread, NULL, worker, NULL);
    pthread_join(thread, NULL);
}
```

The cast is specific to glibc, which currently defines `pthread_t` as an `unsigned long`. POSIX deliberately leaves the type opaque. On this machine the result looked like this:

```
main:   pid=4242 tid=4242 pthread=0x7f7bb8c67740
worker: pid=4242 tid=4243 pthread=0x7f7bb8c666c0
```

Both lines have the same result from `getpid()`, but different results from `gettid()` and `pthread_self()`.

`getpid()` returns the thread-group ID, or TGID, that userspace normally calls the process ID. `gettid()` returns the kernel ID of this particular schedulable task. The main thread has the special property that its TID equals the TGID. The second thread gets its own TID while remaining in the same thread group.

`pthread_t` is neither of those IDs. In glibc it is the address of glibc's `struct pthread`, also called the thread descriptor or TCB. It identifies the userspace bookkeeping for the thread. `pthread_self()` is almost comically short:

```
return (pthread_t) THREAD_SELF;
```

On x86-64, `THREAD_SELF` reads the self pointer through the `fs` segment. The number printed as `pthread_t` is therefore an address in the process, not the integer PID understood by the kernel. It can be reused after a thread has been joined, and portable code compares it with `pthread_equal()` rather than assuming anything about its representation.

So the first map is:

```
userspace handle       pthread_t  ----> glibc struct pthread / TCB
kernel identity        TID        ----> one struct task_struct
process identity       PID/TGID   ----> leader of a task thread group
hardware execution     CPU        ----> runs one selected task context
```

The four values cooperate. They are not aliases.

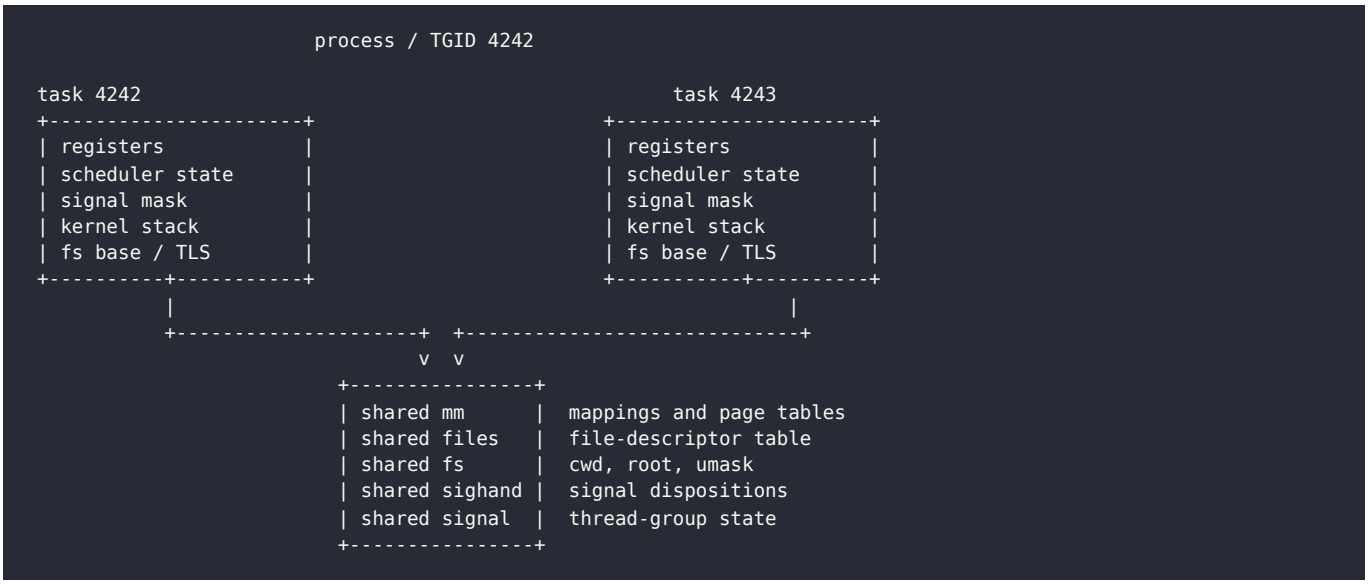
the kernel has tasks, not our nouns

linux avoids deciding whether something is philosophically a process or a thread. The scheduler sees a `struct task_struct`. There is one for the main thread, one for every additional POSIX thread, and one for every ordinary kernel thread.

The structure contains private scheduling and CPU state, then points at the resources which may or may not be shared:

```
struct task_struct {
    struct mm_struct *mm;
    struct mm_struct *active_mm;
    pid_t pid;
    pid_t tgid;
    struct fs_struct *fs;
    struct files_struct *files;
    struct signal_struct *signal;
    struct sighand_struct *sighand;
    struct thread_struct thread;
};
```

That is a heavily reduced sketch of the real structure, but it exposes the trick. A process is not one enormous object which contains a separate species called threads. It is a group of tasks whose pointers lead to the same set of resources.



Each thread also uses a different userspace stack, but those stacks are mappings inside the shared `mm`. They are private by convention and ownership, not by page-table protection. A sufficiently reckless sibling can write into another thread's stack.

This pointer-sharing model is what `clone()` and `clone3()` configure. The flags are less a declaration of "make thread" than a resource-sharing order:

- `CLONE_VM` shares the address space
- `CLONE_FS` shares the working directory, root, and umask state
- `CLONE_FILES` shares the file-descriptor table
- `CLONE_SIGHAND` shares signal dispositions
- `CLONE_THREAD` puts the new task in the same thread group
- `CLONE_SETTLS` installs a new thread-local-storage pointer
- `CLONE_PARENT_SETTID` publishes the new TID to the creator
- `CLONE_CHILD_CLEARTID` clears a userspace word and wakes a futex at exit

The kernel enforces some combinations. `CLONE_THREAD` requires `CLONE_SIGHAND`, and sharing signal handlers requires `CLONE_VM`. A thread group which disagreed about the address holding its signal handler would be a fairly inventive way to ruin an afternoon.

fork, exec, and lightweight processes

`fork()` also creates a new `task_struct`, but it does not pass this sharing set. The child receives a new `mm_struct` with copy-on-write mappings, a new file-descriptor table pointing at the same open file descriptions, and a new thread group. In a multithreaded process, only the calling thread is copied into the child.

`execve()` does not create a process at all. It replaces the calling task's program image, address space, and userspace execution state. If the caller was part of a multithreaded group, the other threads are removed during the exec. The surviving task continues under the process identity with a new program.

The term *lightweight process*, or LWP, comes from systems which described the schedulable kernel entity between a process and a userspace thread. linux's NPTL implementation is one-to-one. Every live `pthread_t` corresponds to one kernel task with one TID. Tools keep

the old vocabulary around. The `LWP` column in `ps` is the TID.

This is not true of every runtime-level task. A coroutine, green thread, or Rust future can be scheduled entirely in userspace. Ten thousand Tokio tasks may move across a small pool of POSIX threads. linux schedules the pool. Tokio schedules the futures on top of it.

There is one more important separation. A userspace thread which makes a system call does not turn into a kernel thread. The same task crosses into ring 0, switches to its kernel stack, performs kernel work, and returns. A kernel thread is a different task which has no ordinary userspace return context at all.

c++, rust, and the same trapdoor

The C program calls `pthread_create()` directly. C++ and Rust add ownership, type erasure, panic or exception policy, and nicer joins, but a GNU/linux build normally reaches the same glibc entry point.

For libstdc++, `std::thread` stores the callable in a heap-allocated `_State` . The current implementation hands that state to its native routine like this:

```
const int err = __gthread_create(&_M_id._M_thread,
                                 &execute_native_thread_routine,
                                 state.get());
```

The POSIX gthread backend defines `__gthread_create` in terms of `pthread_create` . The new thread takes ownership of the state and invokes its virtual `_M_run()` method. `std::thread::join()` goes back through `__gthread_join` , which resolves to `pthread_join()` .

Rust's standard library is even more direct on Unix. Its platform `Thread` contains a `libc::pthread_t` , boxes the Rust startup closure, and performs:

```
let data = Box::into_raw(data);
let ret = libc::pthread_create(
    &mut native, attr.as_ptr(), thread_start, data as *mut _
);
```

The `extern "C"` function `thread_start` reconstructs the box, installs Rust's thread information and stack-overflow handler, then runs the closure. Joining calls `libc::pthread_join` .

Tokio adds another loom above this one. Its blocking pool creates workers with `std::thread::Builder` , optionally sets a stack size, and calls `builder.spawn(...)` . A Tokio future is not a `pthread_t` . The runtime worker executing it is.

```
C pthread_create ------------------------------+
                                               |
C++ std::thread -> libstdc++ __gthread_create --+-> glibc pthread_create
                                               |
Rust std::thread -> libc::pthread_create -------+
                                               |
Tokio task -> Rust worker thread ---------------+
```

The language layers decide how work is represented. glibc decides how a POSIX thread is built. linux receives a clone request.

what glibc builds before clone3

`pthread_create()` first allocates or reuses a userspace stack. Current glibc uses an anonymous `MAP_STACK` mapping and installs a guard area, with a fallback to `PROT_NONE` when the newer guard advice is unavailable. The thread descriptor and static TLS live at an architecture-dependent end of that allocation.

This is the first of two stacks. It is the stack on which the C, C++, or Rust start routine will run in ring 3. The kernel separately allocates a small kernel stack for the new task.

glibc stores the start routine, argument, scheduling attributes, signal mask, TLS metadata, robust-mutex list, rseq area, and join state in `struct pthread`. It returns that descriptor's address as the `pthread_t`:

```
pd->start_routine = start_routine;
pd->arg = arg;
*newthread = (pthread_t) pd;
```

It then prepares the clone flags:

```
CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM |
CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS |
CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID
```

The current `clone_args` gives the new userspace stack to the kernel, puts the thread pointer in `tls`, asks the kernel to write the TID into `pd->tid`, and uses `pd->joinstate` as the clear-and-wake word for thread exit. The essential route is:

```
pthread_create
  -> allocate_stack
  -> initialize struct pthread
  -> __clone_internal
       -> clone3 when available
       -> legacy clone fallback on ENOSYS
  -> start_thread in the child
  -> user start_routine(arg)
```

A trace makes the boundary visible. The exact addresses vary:

```
$ strace -f -e trace=clone,clone3,futex ./needle
clone3({flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|
        CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
        CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, ...}, 88) = 4243
[pid 4242] futex(0x7f7bb8c66..., FUTEX_WAIT_BITSET, ...) = 0
```

On a system where `clone3` is unavailable or blocked, glibc retries through legacy `clone`. That changes the syscall spelling, not the task model.

the child returns somewhere else

`clone3` is unusual because the parent and child continue from the same syscall with different return values. The parent receives the new TID. The child receives zero.

On x86-64, glibc's assembly wrapper preserves its function and argument, executes `syscall`, and branches on `rax`. Written here in Intel syntax, the shape is:

```
mov    eax, __NR_clone3
syscall
test   rax, rax
js     error
jnz    parent_return

xor    ebp, ebp
mov    rdi, r8
call   rdx            # start_thread(pd)
mov    edi, eax
mov    eax, __NR_exit
syscall
```

The syscall entry itself follows the route from crime and strace. Inside `sys_clone3`, linux copies and validates the userspace `clone_args`, then calls `kernel_clone()` and `copy_process()`.

`copy_process()` allocates a `task_struct` and kernel stack, applies the clone flags through helpers such as `copy_mm`, `copy_files`, and `copy_sighand`, allocates a PID, and links the task into its thread group. The decisive identity code is small:

```
if (clone_flags & CLONE_THREAD) {
    p->group_leader = current->group_leader;
    p->tgid = current->tgid;
} else {
    p->group_leader = p;
    p->tgid = p->pid;
}
```

The x86 `copy_thread()` copies the parent's saved userspace registers, writes zero to the child's saved `rax`, replaces the saved userspace `rsp` with the new stack pointer, and handles `CLONE_SETTLS`:

```
*childregs = *current_pt_regs();
childregs->ax = 0;
childregs->sp = sp;
ret = set_new_tls(p, tls);
```

For a 64-bit task, `set_new_tls()` stores that TLS address as the child's FS base. When the child first returns to ring 3, `fs:` already points at its own TCB. `pthread_self()`, `errno`, C `thread_local`, C++ `thread_local`, and Rust thread-local storage can all find the correct instance without asking the kernel for the TID.

Finally, `wake_up_new_task()` makes the new task runnable. It may execute on another CPU before `pthread_create()` has returned in the parent. glibc's startup locks and descriptor ownership rules exist because this race is not a corner case. It is the advertised product.

boot has to invent the first parents

Ordinary tasks are copied from an existing task, which leaves boot with a small chicken-and-egg problem. The first task is not created by `fork()`. `init_task` is a statically initialized `task_struct` compiled into the kernel. It starts with PID 0, `PF_KTHREAD`, `mm = NULL`, and the boot stack. It becomes the idle task for the boot CPU, commonly shown as `swapper/0`.

My old notes put `arch_call_rest_init()` between `start_kernel()` and `rest_init()`. That wrapper is gone from the current generic path. In linux 7.1, `start_kernel()` calls `rest_init()` directly.

The central part is still recognizable:

```
pid = user_mode_thread(kernel_init, NULL, CLONE_FS);
pid = kernel_thread(kthreadd, NULL, NULL, CLONE_FS | CLONE_FILES);
```

The ordering reserves PID 1 for init and PID 2 for `kthreadd`. The first task starts in the kernel function `kernel_init`, waits until `kthreadd` is ready, finishes system initialization, and eventually uses `kernel_execve()` to run `/sbin/init` or one of its fallbacks. That exec supplies the userspace address space and register image from which PID 1 returns to ring 3.

The second task remains in the kernel as `kthreadd`. Once it is ready, `rest_init()` completes `kthreadd_done`, schedules at least once, and turns the original PID 0 task into the CPU idle loop.

```
                        statically built init_task
                                PID 0
                                  |
                  +---------------+---------------+
                  |                               |
                  v                               v
          user_mode_thread                  kernel_thread
             kernel_init                      kthreadd
                PID 1                          PID 2
                  |                               |
            kernel_execve                   creates classic
                  |                          kernel threads
                  v
            /sbin/init in ring 3

       original PID 0 continues as the boot CPU's idle task
```

Calling PID 0 "the kernel process" is understandable shorthand, but slightly misleading. It is the primordial task and later an idle task. The kernel as a whole is not a process living under PID 0.

kthreadd is a request desk

`kthreadd` does not continuously enumerate every kernel thread. It sleeps on a creation queue. A subsystem calling `kthread_create()` allocates a request, adds it to `kthread_create_list`, wakes PID 2, and waits for a completion.

The loop in current `kernel/kthread.c` is essentially:

```
for (;;) {
    if (list_empty(&kthread_create_list))
        schedule();
    while (!list_empty(&kthread_create_list))
        create_kthread(next_request());
}
```

`create_kthread()` calls `kernel_thread()` from the clean context of `kthreadd`. The new child is marked `PF_KTHREAD`, receives its own `task_struct` and kernel stack, and starts in the internal `kthread()` trampoline. It reports successful creation and immediately schedules out in `TASK_UNINTERRUPTIBLE` state.

That stopped start is intentional. The caller may still need to bind the task to a CPU, set NUMA affinity, or finish publishing data. `kthread_create()` only creates. `wake_up_process()` starts. The `kthread_run()` macro performs both.

A minimal kernel-side worker has the familiar shape:

```
static int needle_worker(void *data)
{
    while (!kthread_should_stop()) {
        wait_event_interruptible(queue_wait,
            work_ready || kthread_should_stop());

        if (kthread_should_stop())
            break;

        consume_work(data);
    }
    return 0;
}

worker = kthread_run(needle_worker, state, "needle-worker");
```

Real code must handle wakeup conditions, errors, module lifetime, and freezing more carefully, but the sleep-check-work loop is common. `kthread_stop()` sets the stop bit, wakes the target, waits for its exit completion, and returns the thread function's result.

Workqueues build a more general system on related machinery. Instead of creating a dedicated kthread for each small job, subsystems queue functions to shared or per-CPU `kworker/*` threads. Other recognizable kernel tasks include RCU workers, migration threads, `ksoftirqd/*`, and memory-reclaim daemons such as `kswapd`.

Most classic dynamically created kernel threads descend from PID 2 because `kthreadd` performs the actual clone. There are exceptions to the neat family tree. Per-CPU idle tasks are created specially, and io_uring can create user workers directly from a userspace task with `PF_USER_WORKER` and `PF_IO_WORKER`. "Everything in brackets is a child of kthreadd" is useful at first contact, not a kernel invariant.

the address space which is not there

The usual definition says that kernel threads have no address space. The useful precise version is that an ordinary kernel thread owns no userspace `mm_struct`. Its `task_struct.mm` is `NULL`.

That does not mean the CPU runs without page tables. CR3 must still select a valid translation tree. During a switch to a kernel thread, the scheduler puts the MMU into lazy mode and gives the incoming task an `active_mm` borrowed from the previous task:

```
if (!next->mm) {
    enter_lazy_tlb(prev->active_mm, next);
    next->active_mm = prev->active_mm;
} else {
    switch_mm_irqs_off(prev->active_mm, next->mm, next);
}
```

The kernel half of the address space is mapped in that active context. The kthread has no right to treat the previous process's user mappings as its own, and normal user-memory access helpers have no task `mm` to work with. Reusing the active page-table context avoids a pointless CR3 switch and TLB disruption for code which only touches kernel addresses.

Specialized code can temporarily adopt an address space with `kthread_use_mm(mm)` and later release it with `kthread_unuse_mm(mm)`. The API updates `mm`, switches the MMU context, and includes barriers required by `membarrier`. This is an explicit borrowing arrangement, not the default.

So the original sentence needs one repair:

```
kernel thread:  mm == NULL, active_mm borrowed, executes only kernel code
user task:      mm owns process mappings, active_mm normally equals mm
```

Kernel threads are also called non-interactive. They have no normal terminal, command line, or userspace event loop, but they can still be controlled indirectly through kernel APIs, sysfs, ioctls, netlink, completions, and wait queues. "No userspace instruction stream" is the sturdier distinction.

two stacks, then the hardware

Every schedulable task needs a kernel stack. A POSIX thread therefore has both the large userspace stack arranged by glibc and the much smaller kernel stack allocated with its `task_struct`. A kernel thread only uses the latter.

The old lecture notes say that linux has an 8 KiB kernel stack. That was a reasonable x86-era number, but it is not the current x86-64 definition. In linux 7.1, `THREAD_SIZE` is four base pages, which is 16 KiB with x86's 4 KiB pages. KASAN doubles the order again. With `CONFIG_VMAP_STACK`, those pages are virtually mapped so guard pages can catch an overflow instead of letting one task quietly scribble into adjacent kernel memory.

The userspace stack is not used while the kernel handles a syscall, page fault, or interrupt for that task. Entry code records the userspace state and moves to the task's kernel stack. This keeps untrusted userspace memory out of the kernel's call chain and lets the task sleep halfway through a system call without losing that chain.

For a newly created task, x86 `copy_thread()` prepares an `inactive_task_frame` whose return address is `ret_from_fork_asm`. For a kernel thread it also plants the function in `rbx` and the argument in `r12`. The first time the scheduler selects it, the assembly passes those values into `ret_from_fork()`:

```
mov    rdi, rax        # previously running task
mov    rsi, rsp        # saved register frame
mov    rdx, rbx        # kernel-thread function
mov    rcx, r12        # function argument
call   ret_from_fork
```

For an ordinary kernel thread, `ret_from_fork()` calls that function and the function does not return to userspace. The unusual `kernel_init` path can instead install a user image with `kernel_execve()` and use this return path to enter it. For a freshly cloned POSIX thread, the function slot is empty. The saved register frame returns to glibc's instruction after `clone3`, with `rax == 0`, on the new userspace stack.

No hardware `pthread_t` register exists. linux constructs these frames in software so that an ordinary scheduler switch can start a task which has never run before.

the scheduler does not care about the adjective

A runnable user thread and a runnable kernel thread enter the same scheduler. Scheduling classes and policy decide which eligible task should run on each logical CPU. The choice may happen because a task blocked, yielded, exhausted its current scheduling opportunity, or because another task woke with a stronger claim to the CPU.

A timer interrupt is one possible trigger. A device interrupt, futex wakeup, or inter-processor interrupt can also make a task runnable and request a reschedule. The switch itself is mostly software. linux does not use x86's old hardware task-switch mechanism for ordinary scheduling.

The central route is:

```
schedule
  -> __schedule
  -> pick_next_task
  -> context_switch(prev, next)
       -> switch_mm_irqs_off
       -> switch_to
            -> __switch_to_asm
            -> __switch_to
```

`context_switch()` deals with the address space first. Switching between two processes may load a different CR3, changing the page-table root used by the MMU. PCID can preserve tagged TLB entries across some switches. Switching between two pthreads in one process usually keeps the same `mm`, so this part can avoid an address-space change entirely.

Then `__switch_to_asm` saves the outgoing task's callee-saved registers and performs the line on which the floor moves:

```
mov    QWORD PTR [rdi + TASK_threadsp], rsp
mov    rsp, QWORD PTR [rsi + TASK_threadsp]
```

`rdi` is the previous task and `rsi` the next one. The first instruction saves the old kernel stack pointer in its `thread_struct`. The second loads the new one. The CPU is now executing on another task's saved kernel call chain. Registers are restored from that stack and execution jumps into `__switch_to()`.

The C half switches the remaining architecture state. Current x86 code handles FPU and vector state, saves and loads FS/GS state, installs TLS descriptors, loads the per-thread FS base, changes PKRU when memory protection keys are in use, updates the per-CPU `current_task`, and records the top of the incoming kernel stack.

The FS-base switch is the hardware end of our original `pthread_t`. Two sibling threads can share every virtual address mapping and still have different values for `errno` because the same `fs:` offset resolves through a different base after the context switch.

CPU caches do not get scrubbed merely because another task runs. Coherent caches let threads on different logical CPUs observe shared memory, while the language memory model, atomic instructions, locks, and kernel barriers decide when those observations are valid. A context switch is not a substitute for a mutex.

sleep, wake, and join

Most threads spend much of their life not running. A task waiting for work sets a sleep state, joins a wait queue, and calls `schedule()`. A producer changes the condition, marks the task runnable, and the scheduler may select it again. The saved kernel stack makes the original `schedule()` eventually return as if it were an unusually patient function call.

POSIX mutexes avoid the kernel when possible. An uncontended lock can be an atomic userspace operation on shared memory. Contention uses a futex so the loser can sleep in the kernel instead of burning a CPU. The same userspace word is the meeting point between library policy and kernel waiting.

Thread exit uses a similar seam. glibc marks the descriptor as exiting, runs TLS destructors, releases thread-local libc state, and invokes the per-thread `exit` syscall. It does not use `exit_group`, because the other threads are supposed to survive.

Because the clone used `CLONE_CHILD_CLEARTID`, linux clears the userspace `joinstate` word during `mm_release()` and performs a futex wake:

```
put_user(0, tsk->clear_child_tid);
do_futex(tsk->clear_child_tid, FUTEX_WAKE, 1, ...);
```

`pthread_join()` waits on that word while the thread is alive. Once the kernel has announced the final exit, glibc collects the return value and frees the TCB and userspace stack. C++ and Rust joins end up waiting on the same event.

The kernel task and the userspace descriptor therefore have deliberately different lifetimes. The task can be gone while the joinable descriptor still holds a return value. Reclaiming the descriptor is the joiner's job.

watching the weave

`ps -ef` is a quick first look, but it collapses userspace thread groups. I prefer a view which shows both PID/TGID and LWP/TID:

```
$ ps -eLo pid,ppid,lwp,nlwp,psr,stat,comm
    PID    PPID     LWP NLWP PSR STAT COMMAND
      2       0       2    1   0 S    kthreadd
   4242    4100    4242    2   7 Sl   needle
   4242    4100    4243    2  11 Sl   needle-worker
```

The exact CPU and state will not sit still for the screenshot. `PSR` is the logical CPU on which the task last ran. `NLWP` is the number of tasks in the thread group.

The procfs layout exposes the same distinction:

```
$ ls /proc/4242/task
4242  4243

$ rg '^(Name|Tgid|Pid|PPid|Threads|Kthread):' /proc/4242/task/4243/status
Name:   needle-worker
Tgid:   4242
Pid:    4243
PPid:   4100
Kthread:        0
Threads:        2
```

For a real kernel thread, `/proc/<tid>/status` reports `Kthread: 1`. The empty command line and square brackets commonly shown by `ps` are useful clues, but the flag is less folkloric. With sufficient permission, `/proc/<tid>/stack` shows where a sleeping kernel thread is blocked.

In `htop`, `K` toggles kernel threads and `H` toggles userspace threads. Those are normally Shift+K and Shift+H. Tree view is useful for seeing the large family under `kthreadd`, as long as the exceptions above remain in mind.

For creation and blocking, `strace -f -e clone,clone3,futex` follows all threads across the userspace boundary. For scheduling rather than syscalls, `perf sched record` followed by `perf sched timehist` shows tasks being woken, scheduled, and switched. The two tools observe different layers of the same cloth.

tied off

A POSIX process is a resource-sharing arrangement. A POSIX thread is a userspace contract implemented by glibc with one linux task. A lightweight process is the historical name for that independently schedulable task. A kernel thread is also a task, but one marked `PF_KTHREAD` with no owned userspace `mm` and no normal path back to ring 3.

They meet at `task_struct` and at the scheduler:

```
pthread_t
  -> glibc TCB, TLS, user stack, start routine, join state
  -> clone3 with a resource-sharing recipe
  -> linux task_struct with its own TID and kernel stack
  -> scheduler run queue
  -> x86 register, stack, FS-base, and optional CR3 switch
  -> one logical CPU executes the task
```

The process is not the thing the CPU runs. The CPU runs a task. The process is the collection of things several tasks agreed to share.

source trail

- linux 7.1 boot task creation
- linux 7.1 statically defined `init_task`
- linux 7.1 task cloning and resource sharing
- linux 7.1 `task_struct`
- linux 7.1 kthread implementation
- linux 7.1 kthread API
- linux 7.1 x86 task setup
- linux 7.1 x86 context-switch assembly
- linux 7.1 scheduler context switch
- linux 7.1 x86-64 kernel-stack size
- glibc 2.44 `pthread_create`
- glibc 2.44 stack allocator
- glibc 2.44 `pthread_t` definition
- glibc 2.44 `pthread_self` and x86 TLS
- glibc 2.44 clone/clone3 selection
- glibc 2.44 x86-64 clone3 trampoline
- glibc 2.44 pthread join path
- libstdc++ `std::thread` implementation
- Rust standard-library Unix threads
- Tokio's native blocking-worker creation
- linux `clone(2)` manual
- linux `pthreads(7)` manual
- the original Caltech kernel-thread lecture notes
- the original kernel-thread notes from Packt
- Intel 64 and IA-32 software developer manuals