Threads and Concurrency

Advanced Operating Systems — Week 4

Hunter College, CUNY

Faye

February 2026

Contents

1 Why Threads?

A **process** is a heavyweight abstraction: it owns an address space, file descriptors, signal handlers, and more. Creating a new process (`fork`) is expensive because the kernel must duplicate or copy-on-write all of these resources. When we want concurrency *within* a single application (e.g., a web server handling many clients), spawning a new process for each task is wasteful.

A **thread** (sometimes called a *lightweight process*) is an independent flow of execution *within* a process. Multiple threads share the same address space and resources, making them cheaper to create and faster to switch between.

Attribute	Process	Thread
Address Space	Own (isolated)	Shared with other threads
Creation Time	Expensive (fork + COW)	Cheap (10–100× faster)
Context Switch	Expensive (TLB flush, etc.)	Cheap (5–10× faster)
Communication	Requires IPC (pipes, shm)	Direct via shared memory
Isolation	Strong (separate address spaces)	Weak (a bug can corrupt shared data)
Crash Impact	Other processes unaffected	Crash kills entire process (all threads)

2 What Threads Share vs. Own

Shared (per-process)

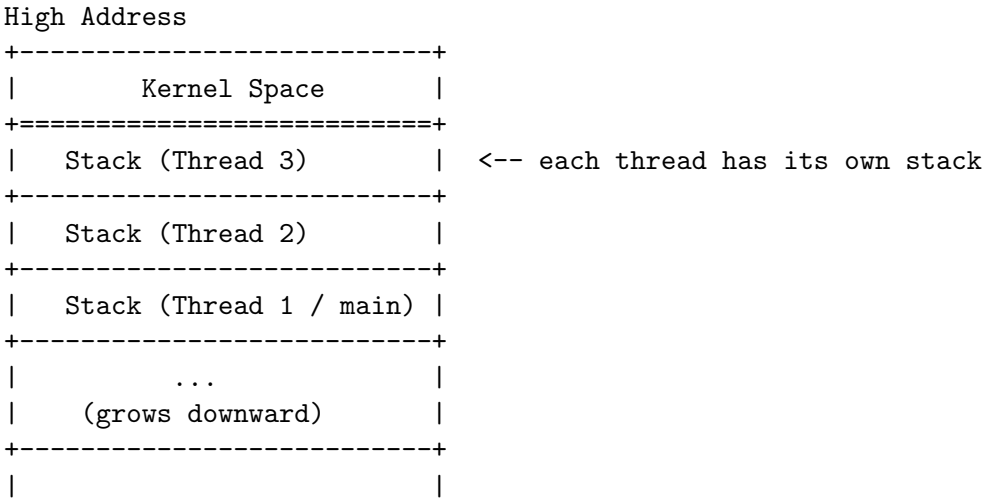
- Code (text segment)
- Global variables (data/BSS)
- Heap
- File descriptors
- Current working directory
- Signal handlers (disposition)

Private (per-thread)

- Thread ID (TID)
- Stack
- Registers (including PC, SP)
- Scheduling priority
- Signal mask
- `errno`

3 Thread Memory Layout

The following diagram shows how multiple threads coexist within a single process address space:



```
|          Heap              |  <-- shared among all threads
|    (grows upward)          |
+----------------------------+
|     BSS (uninitialized)    |  <-- shared
+----------------------------+
|     Data (initialized)     |  <-- shared
+----------------------------+
|     Text (code)            |  <-- shared
+----------------------------+
Low Address
```

Each thread's stack is allocated in a separate region (typically 2–8 MB each, configurable). Stack overflow in one thread can corrupt another thread's memory — there is no hardware-enforced guard between thread stacks (though guard pages help).

4 Benefits of Threads

1. **Responsiveness:** In a GUI application, a worker thread can perform computation while the UI thread remains responsive to user input.
2. **Resource Sharing:** Threads within a process share code, data, and files by default — no need for explicit IPC setup.
3. **Economy:** Thread creation is 10–100× cheaper than process creation. Context switching between threads is 5–10× faster than between processes (no TLB flush, no address-space switch).
4. **Scalability:** On multicore/multiprocessor systems, threads can run in true parallel on different cores, achieving speedup proportional to the number of cores (limited by Amdahl's law).

5 Challenges of Multithreading

- **Race conditions:** Two threads accessing shared data concurrently can produce inconsistent results depending on interleaving.
- **Deadlocks:** Two or more threads each waiting for a resource held by the other, resulting in permanent blocking.
- **Heisenbugs:** Concurrency bugs that change behavior when you try to observe/debug them (e.g., adding a `printf` changes timing enough to hide the bug).

Race Condition: counter++ Interleaving

Suppose two threads both execute `counter++` where `counter` starts at 5:

Step	Thread A	Step	Thread B
1	`LOAD counter` → reg = 5		
		2	`LOAD counter` → reg = 5
3	`ADD 1` → reg = 6		
		4	`ADD 1` → reg = 6
5	`STORE reg` → counter = 6		
		6	`STORE reg` → counter = 6

Expected: counter = 7. **Actual:** counter = 6. One increment was lost.

6 Thread Models

The relationship between **user-level threads** (managed by a threading library) and **kernel-level threads** (managed by the OS) defines the threading model.

6.1 Many-to-One (N:1)

Many user threads map to a single kernel thread. The thread library manages scheduling entirely in user space.

- **Pros:** Very fast thread creation and switching (no system calls). Portable.
- **Cons:** A blocking system call blocks *all* threads. Cannot exploit multiple cores.
- **Examples:** GNU Portable Threads, early Green Threads (Solaris), Ruby < 1.9.

6.2 One-to-One (1:1)

Each user thread maps to a dedicated kernel thread.

- **Pros:** True parallelism on multicore. One thread blocking does not affect others.
- **Cons:** Higher overhead for creation and switching (each requires a kernel call). OS may limit the number of kernel threads.
- **Examples:** Linux (NPTL), Windows, modern macOS.

6.3 Many-to-Many (M:N)

M user threads multiplexed onto N kernel threads ($M \geq N$).

- **Pros:** Combines benefits — true parallelism with lower overhead than 1:1.
- **Cons:** Very complex to implement. Scheduling interaction between user-level and kernel-level schedulers is tricky.
- **Examples:** Solaris LWP, Go goroutines (M:N with its own scheduler), older IRIX.

6.4 Comparison Table

Aspect	N:1	1:1	M:N
Parallelism	No	Yes	Yes
Blocking call	Blocks all	Blocks one	Blocks one
Creation cost	Very low	Moderate	Low–moderate
Complexity	Low	Low	High
Modern usage	Rare	Dominant (Linux, Windows)	Go, some specialized

Modern Trend

Most modern systems use the **1:1 model** because multicore CPUs are universal and the overhead of kernel thread management is acceptable. The M:N model has seen a resurgence with language runtimes like **Go** (goroutines) and **Erlang** (lightweight processes) that implement their own user-space schedulers atop kernel threads.

7 Linux NPTL and `clone()`

In Linux, both processes and threads are created by the `clone()` system call. The difference is which resources are **shared** vs. **copied**:

- `CLONE_VM` — share the virtual memory (address space).
- `CLONE_FS` — share filesystem information (root, cwd, umask).

- `CLONE_FILES` — share the file descriptor table.
- `CLONE_SIGHAND` — share signal handlers.
- `CLONE_THREAD` — place in the same thread group (share PID, appear as one process).

`fork()` calls `clone()` with *none* of these flags (everything copied).
`pthread_create()` calls `clone()` with *all* of these flags (everything shared).

The **Native POSIX Threads Library (NPTL)** is the standard threading implementation on Linux since glibc 2.3.2 (2003). It uses the 1:1 model, with each pthread mapping to exactly one kernel task (schedulable entity).

8 POSIX Threads (pthreads)

8.1 `pthread_create`

Listing 1: pthread_create signature

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

- `thread`: output parameter — receives the new thread's ID.
- `attr`: thread attributes (`NULL` for defaults).
- `start_routine`: function pointer — the thread entry point.
- `arg`: argument passed to `start_routine`.
- Returns 0 on success, error number on failure.

Listing 2: Basic thread creation

```
#include <stdio.h>
#include <pthread.h>

void *greet(void *arg) {
    char *name = (char *)arg;
    printf("Hello from thread: %s\n", name);
    return NULL;
}

int main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, greet, "Worker");
    pthread_join(tid, NULL);
    printf("Thread finished.\n");
    return 0;
}
```

8.2 `pthread_join` and Return Values

`pthread_join(tid, &retval)` blocks until thread `tid` terminates, and optionally retrieves its return value.

Listing 3: Returning a value from a thread

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```c
void *compute(void *arg) {
    int input = *(int *)arg;
    int *result = malloc(sizeof(int));
    *result = input * input;
    return result;
}

int main(void) {
    int value = 7;
    pthread_t tid;
    pthread_create(&tid, NULL, compute, &value);

    void *retval;
    pthread_join(tid, &retval);
    printf("Result: %d\n", *(int *)retval);  // prints 49
    free(retval);
    return 0;
}
```

8.3 pthread_exit vs. return

- `return value;` from the thread function is equivalent to `pthread_exit(value)`.
- `pthread_exit()` can be called from *any* function within the thread, not just the start routine. It terminates only the calling thread.
- Calling `exit()` from any thread terminates the **entire process**.
- If `main()` calls `pthread_exit()` instead of `return`, other threads continue running.

8.4 pthread_detach

A detached thread's resources are automatically reclaimed when it terminates (no `pthread_join` needed or allowed).

```c
pthread_t tid;
pthread_create(&tid, NULL, worker, NULL);
pthread_detach(tid);  // no join needed; resources freed on exit
```

8.5 Creating Multiple Threads

Listing 4: Creating multiple threads

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 5

void *worker(void *arg) {
    int id = *(int *)arg;
    printf("Thread %d running\n", id);
    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
```

```
    int ids[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, worker, &ids[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
```

Common Pitfall: Passing `&i` Directly

A frequent mistake is passing the address of the loop variable directly:

```
// WRONG: all threads may see the same value of i
for (int i = 0; i < NUM_THREADS; i++) {
    pthread_create(&threads[i], NULL, worker, &i);
}
```

Because `i` is modified on each iteration, threads may read a stale or incorrect value. The variable is shared and the read is unsynchronized.

Correct solution: use a separate array so each thread gets its own copy:

```
int ids[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++) {
    ids[i] = i;
    pthread_create(&threads[i], NULL, worker, &ids[i]);
}
```

Alternative: cast the integer to `void*` (works when `sizeof(int) <= sizeof(void*)`):

```
pthread_create(&threads[i], NULL, worker, (void *)(intptr_t)i);
// In worker: int id = (int)(intptr_t)arg;
```

9 Thread Attributes

Thread behavior can be configured via `pthread_attr_t`:

Attribute	Default	Description
Detach state	`JOINABLE`	Whether the thread must be joined or is auto-cleaned.
Stack size	System default (2–8 MB)	Size of the thread's stack.
Stack address	System-chosen	Specify a custom stack location.
Scheduling policy	`SCHED_OTHER`	Scheduling policy (`FIFO`, `RR`, `OTHER`).
Scheduling priority	0	Priority within the chosen policy.
Inherit scheduler	`INHERIT`	Inherit scheduling attributes from the creating thread.
Scope	`SYSTEM`	Contention scope (`SYSTEM` = kernel-level, `PROCESS` = user-level).
Guard size	Page size (4 KB)	Size of guard region at the end of the stack.

Listing 5: Setting thread attributes

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_attr_setstacksize(&attr, 1024 * 1024);  // 1 MB stack

pthread_t tid;
pthread_create(&tid, &attr, worker, NULL);
pthread_attr_destroy(&attr);
```

10 Thread-Specific Data (TSD)

Sometimes threads need "global" variables that are **private to each thread**. POSIX provides thread-specific data via keys:

Listing 6: Thread-specific data example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

static pthread_key_t key;

void destructor(void *value) {
    free(value);  // automatically called when thread exits
}

void *worker(void *arg) {
    int *data = malloc(sizeof(int));
    *data = *(int *)arg;
    pthread_setspecific(key, data);

    // Later, retrieve thread-specific data
    int *my_data = pthread_getspecific(key);
    printf("Thread %d: my_data = %d\n", *my_data, *my_data);
    return NULL;
}

int main(void) {
    pthread_key_create(&key, destructor);

    pthread_t t1, t2;
    int id1 = 1, id2 = 2;
    pthread_create(&t1, NULL, worker, &id1);
    pthread_create(&t2, NULL, worker, &id2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_key_delete(key);
    return 0;
}
```

Modern Alternative: `__thread` / `thread_local`

C11 provides the `_Thread_local` (or `thread_local` with `<threads.h>`) storage class specifier, and GCC/Clang support `__thread`. These are simpler than the POSIX key API:

```
__thread int my_errno;  // each thread gets its own copy
```

11 Synchronization Preview

11.1 Why `counter++` Is Not Atomic

The C statement `counter++` compiles to multiple machine instructions:

Listing 7: counter++ in assembly (x86)

```
    LOAD  counter -> register      ; read from memory
    ADD   1       -> register      ; increment in register
    STORE register -> counter      ; write back to memory
```

If two threads execute this concurrently, their instructions can interleave, causing a **lost update** (as shown in the race condition example in Section 5).

11.2 Critical Section Requirements

A **critical section** is a code segment that accesses shared resources. A correct solution to the critical section problem must satisfy:

1. **Mutual Exclusion:** At most one thread is in the critical section at any time.
2. **Progress:** If no thread is in the critical section and some threads wish to enter, one of them must be allowed to enter (no unnecessary blocking).
3. **Bounded Waiting:** There is a bound on the number of times other threads can enter the critical section after a thread has requested entry (no starvation).

11.3 Mutex Example

A **mutex** (mutual exclusion lock) is the most basic synchronization primitive:

Listing 8: Protecting a shared counter with a mutex

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;              // critical section
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);
```

```
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d\n", counter);  // always 2000000
    return 0;
}
```

12 Thread Pools

12.1 Motivation

Creating a new thread for every task has problems:

- Thread creation/destruction overhead adds up for short-lived tasks.
- Unbounded thread creation can exhaust system resources (memory, kernel structures).
- No control over the degree of parallelism.

12.2 Architecture

A **thread pool** pre-creates a fixed number of worker threads. Tasks are submitted to a **work queue**. Idle workers dequeue and execute tasks. When a worker finishes, it returns to the pool and picks up the next task.

Thread Pool Sizing

The ideal number of threads depends on the workload:

$$N_{\text{threads}} = N_{\text{cores}} \times \left(1 + \frac{W}{C}\right)$$

where W = average time a task spends **waiting** (I/O, network) and C = average time a task spends **computing** (CPU).

- **CPU-bound** tasks ($W/C \approx 0$): $N_{\text{threads}} \approx N_{\text{cores}}$ (or $N_{\text{cores}} + 1$ to cover occasional page faults).
- **I/O-bound** tasks ($W/C \gg 1$): many more threads than cores, e.g., a web server with $W/C = 9$ on a 4-core machine: $4 \times (1 + 9) = 40$ threads.

13 Common Pitfalls

Common Multithreading Pitfalls

1. **Passing stack variables to threads:** The variable may go out of scope or be modified before the thread reads it. Always use heap-allocated or per-element array storage.
2. **Forgetting to join or detach:** A joinable thread that is never joined leaks resources (like a zombie process). Always either `pthread_join` or `pthread_detach`.
3. **Accessing shared data without synchronization:** Even "harmless" reads can see torn values on some architectures. Always protect shared state with mutexes or atomics.
4. **Holding locks across blocking calls:** If a thread holds a mutex and then blocks on I/O, all other threads waiting for that mutex are stalled. Keep critical sections short.
5. **Calling `exit()` from a thread:** This terminates the *entire process*. Use `pthread_exit()` to terminate only the calling thread.
6. **Assuming execution order:** Thread scheduling is non-deterministic. Never rely on threads executing in a particular order without explicit synchronization.

14 Key Takeaways

Key Takeaways — Threads and Concurrency

1. Threads provide concurrency within a process with **lower overhead** than separate processes: 10–100× cheaper creation, 5–10× faster context switch.
2. Threads **share** code, data, heap, and file descriptors, but each has its own **stack**, registers, TID, and signal mask.
3. The **1:1 threading model** (one user thread = one kernel thread) dominates modern systems (Linux NPTL, Windows). The **M:N model** is used by language runtimes (Go, Erlang).
4. The `counter++` problem illustrates why shared data must be protected. A correct critical-section solution requires **mutual exclusion**, **progress**, and **bounded waiting**.
5. **Thread pools** control resource usage and amortize creation cost. Size them with $N_{\text{threads}} = N_{\text{cores}} \times (1 + W/C)$.