zerocopy

The `zerocopy` crate (from Fuchsia) provides traits and macros for safely converting between byte sequences and other types.

```
use zerocopy::{Immutable, IntoBytes};

#[repr(u32)]
#[derive(Debug, Default, Immutable, IntoBytes)]
enum RequestType {
    #[default]
    In = 0,
    Out = 1,
    Flush = 4,
}

#[repr(C)]
#[derive(Debug, Default, Immutable, IntoBytes)]
struct VirtioBlockRequest {
    request_type: RequestType,
    reserved: u32,
    sector: u64,
}

fn main() {
    let request = VirtioBlockRequest {
        request_type: RequestType::Flush,
        sector: 42,
        ..Default::default()
    };

    assert_eq!(
        request.as_bytes(),
        &[4, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0]
    );
}
```

This is not suitable for MMIO (as it doesn't use volatile reads and writes), but can be useful for working with structures shared with hardware e.g. by DMA, or sent over some external interface.

▼ *Speaker Notes*

- `FromBytes` can be implemented for types for which any byte pattern is valid, and so can safely be converted from an untrusted sequence of bytes.
- Attempting to derive `FromBytes` for these types would fail, because `RequestType` doesn't use all possible u32 values as discriminants, so not all byte patterns are valid.
- `zerocopy::byteorder` has types for byte-order aware numeric primitives.
- Run the example with `cargo run` under `src/bare-metal/useful-crates/zerocopy-example/`. (It won't run in the Playground because of the crate dependency.)

aarch64-paging

The `aarch64-paging` crate lets you create page tables according to the AArch64 Virtual Memory System Architecture.

```
use aarch64_paging::{
    idmap::IdMap,
    paging::{Attributes, MemoryRegion},
};

const ASID: usize = 1;
const ROOT_LEVEL: usize = 1;

// Create a new page table with identity mapping.
let mut idmap = IdMap::new(ASID, ROOT_LEVEL);
// Map a 2 MiB region of memory as read-only.
idmap.map_range(
    &MemoryRegion::new(0x80200000, 0x80400000),
    Attributes::NORMAL | Attributes::NON_GLOBAL | Attributes::READ_ONLY,
).unwrap();
// Set `TTBR0_EL1` to activate the page table.
idmap.activate();
```

▼ *Speaker Notes*

- This is used in Android for the Protected VM Firmware.
- There's no easy way to run this example by itself, as it needs to run on real hardware or under QEMU.

buddy_system_allocator

`buddy_system_allocator` is a crate implementing a basic buddy system allocator. It can be used both to implement `GlobalAlloc` (using `LockedHeap`) so you can use the standard `alloc` crate (as we saw before), or for allocating other address space (using `FrameAllocator`). For example, we might want to allocate MMIO space for PCI BARs:

```
use buddy_system_allocator::FrameAllocator;
use core::alloc::Layout;

fn main() {
    let mut allocator = FrameAllocator::<32>::new();
    allocator.add_frame(0x200_0000, 0x400_0000);

    let layout = Layout::from_size_align(0x100, 0x100).unwrap();
    let bar = allocator
        .alloc_aligned(layout)
        .expect("Failed to allocate 0x100 byte MMIO region");
    println!("Allocated 0x100 byte MMIO region at {:#x}", bar);
}
```

▼ *Speaker Notes*

- PCI BARs always have alignment equal to their size.
- Run the example with `cargo run` under `src/bare-metal/useful-crates/allocator-example/`. (It won't run in the Playground because of the crate dependency.)

tinyvec

Sometimes you want something which can be resized like a `Vec`, but without heap allocation. `tinyvec` provides this: a vector backed by an array or slice, which could be statically allocated or on the stack, which keeps track of how many elements are used and panics if you try to use more than are allocated.

```
use tinyvec::{ArrayVec, array_vec};

fn main() {
    let mut numbers: ArrayVec<[u32; 5]> = array_vec!(42, 66);
    println!("{numbers:?}");
    numbers.push(7);
    println!("{numbers:?}");
    numbers.remove(1);
    println!("{numbers:?}");
}
```

▼ *Speaker Notes*

- `tinyvec` requires that the element type implement `Default` for initialisation.
- The Rust Playground includes `tinyvec`, so this example will run fine inline.

spin

`std::sync::Mutex` and the other synchronisation primitives from `std::sync` are not available in `core` or `alloc`. How can we manage synchronisation or interior mutability, such as for sharing state between different CPUs?

The `spin` crate provides spinlock-based equivalents of many of these primitives.

```
use spin::mutex::SpinMutex;

static COUNTER: SpinMutex<u32> = SpinMutex::new(0);

fn main() {
    dbg!(COUNTER.lock());
    *COUNTER.lock() += 2;
    dbg!(COUNTER.lock());
}
```

▼ *Speaker Notes*

- Be careful to avoid deadlock if you take locks in interrupt handlers.
- `spin` also has a ticket lock mutex implementation; equivalents of `RwLock`, `Barrier` and `Once` from `std::sync`; and `Lazy` for lazy initialisation.
- The `once_cell` crate also has some useful types for late initialisation with a slightly different approach to `spin::once::Once`.
- The Rust Playground includes `spin`, so this example will run fine inline.

Bare-Metal on Android

To build a bare-metal Rust binary in AOSP, you need to use a `rust_ffi_static` Soong rule to build your Rust code, then a `cc_binary` with a linker script to produce the binary itself, and then a `raw_binary` to convert the ELF to a raw binary ready to be run.

```
rust_ffi_static {
    name: "libvmbase_example",
    defaults: ["vmbase_ffi_defaults"],
    crate_name: "vmbase_example",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libvmbase",
    ],
}

cc_binary {
    name: "vmbase_example",
    defaults: ["vmbase_elf_defaults"],
    srcs: [
        "idmap.S",
    ],
    static_libs: [
        "libvmbase_example",
    ],
    linker_scripts: [
        "image.ld",
        ":vmbase_sections",
    ],
}

raw_binary {
    name: "vmbase_example_bin",
    stem: "vmbase_example.bin",
    src: ":vmbase_example",
    enabled: false,
    target: {
        android_arm64: {
            enabled: true,
        },
    },
}
```

vmbase

For VMs running under crosvm on aarch64, the vmbase library provides a linker script and useful defaults for the build rules, along with an entry point, UART console logging and more.

```
#![no_main]
#![no_std]

use vmbase::{main, println};

main!(main);

pub fn main(arg0: u64, arg1: u64, arg2: u64, arg3: u64) {
    println!("Hello world");
}
```

▼ *Speaker Notes*

- The `main!` macro marks your main function, to be called from the `vmbase` entry point.
- The `vmbase` entry point handles console initialisation, and issues a PSCI_SYSTEM_OFF to shutdown the VM if your main function returns.

Exercises

We will write a driver for the PL031 real-time clock device.

▼ *Speaker Notes*

After looking at the exercises, you can look at the solutions provided.

RTC driver

The QEMU aarch64 virt machine has a PL031 real-time clock at 0x9010000. For this exercise, you should write a driver for it.

1. Use it to print the current time to the serial console. You can use the `chrono` crate for date/time formatting.
2. Use the match register and raw interrupt status to busy-wait until a given time, e.g. 3 seconds in the future. (Call `core::hint::spin_loop` inside the loop.)
3. *Extension if you have time:* Enable and handle the interrupt generated by the RTC match. You can use the driver provided in the `arm-gic` crate to configure the Arm Generic Interrupt Controller.
    - Use the RTC interrupt, which is wired to the GIC as `IntId::spi(2)`.
    - Once the interrupt is enabled, you can put the core to sleep via `arm_gic::wfi()`, which will cause the core to sleep until it receives an interrupt.

Download the exercise template and look in the `rtc` directory for the following files.

src/main.rs:

```rust
#![no_main]
#![no_std]

mod exceptions;
mod logger;

use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::GicV3;
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut Gicd = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut GicrSgi = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
});

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // SAFETY: `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic =
        unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS, 1, false) };
    gic.setup(0);

    // TODO: Create instance of RTC driver and print current time.

    // TODO: Wait for 3 seconds.

    system_off::<Hvc>().unwrap();
```

```
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

src/exceptions.rs (you should only need to change this for the 3rd part of the exercise):

```rust
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use arm_gic::gicv3::{GicV3, InterruptGroup};
use log::{error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    trace!("irq_current");
    let intid = GicV3::get_and_acknowledge_interrupt(InterruptGroup::Group1)
        .expect("No pending interrupt");
    info!("IRQ {intid:?}");
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}
```

```
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

src/logger.rs (you shouldn't need to change this):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use arm_pl011_uart::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart<'static>>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(
    uart: Uart<'static>,
    max_level: LevelFilter,
) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

Cargo.toml (you shouldn't need to change this):

```
[workspace]

[package]
name = "rtc"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
aarch64-paging = { version = "0.10.0", default-features = false }
aarch64-rt = "0.2.2"
arm-gic = "0.6.0"
arm-pl011-uart = "0.3.1"
bitflags = "2.9.1"
chrono = { version = "0.4.41", default-features = false }
log = "0.4.27"
safe-mmio = "0.2.5"
smccc = "0.2.2"
spin = "0.10.0"
zerocopy = "0.8.26"
```

build.rs (you shouldn't need to change this):

```
// Copyright 2025 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

fn main() {
    println!("cargo:rustc-link-arg=-Timage.ld");
    println!("cargo:rustc-link-arg=-Tmemory.ld");
    println!("cargo:rerun-if-changed=memory.ld");
}
```

memory.ld (you shouldn't need to change this):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

MEMORY
{
        image : ORIGIN = 0x40080000, LENGTH = 2M
}
```

Makefile (you shouldn't need to change this):

```
# Copyright 2023 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

.PHONY: build qemu_minimal qemu qemu_logger

all: rtc.bin

build:
    cargo build

rtc.bin: build
    cargo objcopy -- -O binary $@

qemu: rtc.bin
    qemu-system-aarch64 -machine virt,gic-version=3 -cpu max -serial mon:stdio -
display none -kernel $< -s

clean:
    cargo clean
    rm -f *.bin
```

.cargo/config.toml (you shouldn't need to change this):

```
[build]
target = "aarch64-unknown-none"
```

Run the code in QEMU with `make qemu`.

Bare Metal Rust Afternoon

RTC driver

(back to exercise)

main.rs:

```rust
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl031;

use crate::pl031::Rtc;
use arm_gic::{IntId, Trigger, irq_enable, wfi};
use chrono::{TimeZone, Utc};
use core::hint::spin_loop;
use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::GicV3;
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut Gicd = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut GicrSgi = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
});

/// Base address of the PL031 RTC.
const PL031_BASE_ADDRESS: NonNull<pl031::Registers> =
    NonNull::new(0x901_0000 as _).unwrap();
/// The IRQ used by the PL031 RTC.
const PL031_IRQ: IntId = IntId::spi(2);

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // SAFETY: `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
```

```
    let mut gic =
        unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS, 1, false) };
    gic.setup(0);

    // SAFETY: `PL031_BASE_ADDRESS` is the base address of a PL031 device, and
    // nothing else accesses that address range.
    let mut rtc = unsafe { Rtc::new(UniqueMmioPointer::new(PL031_BASE_ADDRESS)) };
    let timestamp = rtc.read();
    let time = Utc.timestamp_opt(timestamp.into(), 0).unwrap();
    info!("RTC: {time}");

    GicV3::set_priority_mask(0xff);
    gic.set_interrupt_priority(PL031_IRQ, None, 0x80);
    gic.set_trigger(PL031_IRQ, None, Trigger::Level);
    irq_enable();
    gic.enable_interrupt(PL031_IRQ, None, true);

    // Wait for 3 seconds, without interrupts.
    let target = timestamp + 3;
    rtc.set_match(target);
    info!("Waiting for {}", Utc.timestamp_opt(target.into(), 0).unwrap());
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.matched() {
        spin_loop();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("Finished waiting");

    // Wait another 3 seconds for an interrupt.
    let target = timestamp + 6;
    info!("Waiting for {}", Utc.timestamp_opt(target.into(), 0).unwrap());
    rtc.set_match(target);
    rtc.clear_interrupt();
    rtc.enable_interrupt(true);
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.interrupt_pending() {
        wfi();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("Finished waiting");

    system_off::<Hvc>().unwrap();
    panic!("system_off returned");
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

pl031.rs:

```rust
#[repr(C, align(4))]
pub struct Registers {
    /// Data register
    dr: ReadPure<u32>,
    /// Match register
    mr: ReadPureWrite<u32>,
    /// Load register
    lr: ReadPureWrite<u32>,
    /// Control register
    cr: ReadPureWrite<u8>,
    _reserved0: [u8; 3],
    /// Interrupt Mask Set or Clear register
    imsc: ReadPureWrite<u8>,
    _reserved1: [u8; 3],
    /// Raw Interrupt Status
    ris: ReadPure<u8>,
    _reserved2: [u8; 3],
    /// Masked Interrupt Status
    mis: ReadPure<u8>,
    _reserved3: [u8; 3],
    /// Interrupt Clear Register
    icr: WriteOnly<u8>,
    _reserved4: [u8; 3],
}

/// Driver for a PL031 real-time clock.
#[derive(Debug)]
pub struct Rtc<'a> {
    registers: UniqueMmioPointer<'a, Registers>,
}

impl<'a> Rtc<'a> {
    /// Constructs a new instance of the RTC driver for a PL031 device with the
    /// given set of registers.
    pub fn new(registers: UniqueMmioPointer<'a, Registers>) -> Self {
        Self { registers }
    }

    /// Reads the current RTC value.
    pub fn read(&self) -> u32 {
        field_shared!(self.registers, dr).read()
    }

    /// Writes a match value. When the RTC value matches this then an interrupt
    /// will be generated (if it is enabled).
    pub fn set_match(&mut self, value: u32) {
        field!(self.registers, mr).write(value);
    }

    /// Returns whether the match register matches the RTC value, whether or not
    /// the interrupt is enabled.
    pub fn matched(&self) -> bool {
        let ris = field_shared!(self.registers, ris).read();
        (ris & 0x01) != 0
    }

    /// Returns whether there is currently an interrupt pending.
    ///
    /// This should be true if and only if `matched` returns true and the
    /// interrupt is masked.
    pub fn interrupt_pending(&self) -> bool {
        let mis = field_shared!(self.registers, mis).read();
        (mis & 0x01) != 0
    }

    /// Sets or clears the interrupt mask.
    ///
    /// When the mask is true the interrupt is enabled; when it is false the
    /// interrupt is disabled.
    pub fn enable_interrupt(&mut self, mask: bool) {
```

```
    }

    /// Clears a pending interrupt, if any.
    pub fn clear_interrupt(&mut self) {
        field!(self.registers, icr).write(0x01);
    }
}
```

Welcome to Concurrency in Rust

Rust has full support for concurrency using OS threads with mutexes and channels.

The Rust type system plays an important role in making many concurrency bugs compile time bugs. This is often referred to as *fearless concurrency* since you can rely on the compiler to ensure correctness at runtime.

Schedule

Including 10 minute breaks, this session should take about 3 hours and 20 minutes. It contains:

Segment	Duration
Threads	30 minutes
Channels	20 minutes
Send and Sync	15 minutes
Shared State	30 minutes
Exercises	1 hour and 10 minutes

▼ *Speaker Notes*

- Rust lets us access OS concurrency toolkit: threads, sync. primitives, etc.
- The type system gives us safety for concurrency without any special features.
- The same tools that help with “concurrent” access in a single thread (e.g., a called function that might mutate an argument or save references to it to read later) save us from multi-threading issues.

Threads

This segment should take about 30 minutes. It contains:

Slide	Duration
Plain Threads	15 minutes
Scoped Threads	15 minutes

Plain Threads

Rust threads work similarly to threads in other languages:

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 0..10 {
            println!("Count in thread: {i}!");
            thread::sleep(Duration::from_millis(5));
        }
    });

    for i in 0..5 {
        println!("Main thread: {i}");
        thread::sleep(Duration::from_millis(5));
    }
}
```

- Spawning new threads does not automatically delay program termination at the end of `main`.
- Thread panics are independent of each other.
  - Panics can carry a payload, which can be unpacked with `Any::downcast_ref`.

▼ *Speaker Notes*

This slide should take about 15 minutes.

- Run the example.
  - 5ms timing is loose enough that main and spawned threads stay mostly in lockstep.
  - Notice that the program ends before the spawned thread reaches 10!
  - This is because `main` ends the program and spawned threads do not make it persist.
    - Compare to `pthreads` /C++ `std::thread` / `boost::thread` if desired.
- How do we wait around for the spawned thread to complete?
- `thread::spawn` returns a `JoinHandle`. Look at the docs.
  - `JoinHandle` has a `.join()` method that blocks.
- Use `let handle = thread::spawn(...)` and later `handle.join()` to wait for the thread to finish and have the program count all the way to 10.
- Now what if we want to return a value?
- Look at docs again:
  - `thread::spawn`'s closure returns `T`
  - `JoinHandle` `.join()` returns `thread::Result<T>`
- Use the `Result` return value from `handle.join()` to get access to the returned value.
- Ok, what about the other case?
  - Trigger a panic in the thread. Note that this doesn't panic `main`.
  - Access the panic payload. This is a good time to talk about `Any`.
- Now we can return values from threads! What about taking inputs?

    - Move it in, see we can compute and then return a derived value.

- If we want to borrow?
    - Main kills child threads when it returns, but another function would just return and leave them running.
    - That would be stack use-after-return, which violates memory safety!
    - How do we avoid this? See next slide.

Scoped Threads

Normal threads cannot borrow from their environment:

```
use std::thread;

fn foo() {
    let s = String::from("Hello");
    thread::spawn(|| {
        dbg!(s.len());
    });
}

fn main() {
    foo();
}
```

However, you can use a scoped thread for this:

```
use std::thread;

fn foo() {
    let s = String::from("Hello");
    thread::scope(|scope| {
        scope.spawn(|| {
            dbg!(s.len());
        });
    });
}

fn main() {
    foo();
}
```

▼ *Speaker Notes*

This slide should take about 13 minutes.

- The reason for that is that when the `thread::scope` function completes, all the threads are guaranteed to be joined, so they can return borrowed data.
- Normal Rust borrowing rules apply: you can either borrow mutably by one thread, or immutably by any number of threads.

Channels

This segment should take about 20 minutes. It contains:

Slide	Duration
Senders and Receivers	10 minutes
Unbounded Channels	2 minutes
Bounded Channels	10 minutes

Senders and Receivers

Rust channels have two parts: a `Sender<T>` and a `Receiver<T>`. The two parts are connected via the channel, but you only see the end-points.

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    tx.send(10).unwrap();
    tx.send(20).unwrap();

    println!("Received: {:?}", rx.recv());
    println!("Received: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("Received: {:?}", rx.recv());
}
```

▼ *Speaker Notes*

This slide should take about 9 minutes.

- `mpsc` stands for Multi-Producer, Single-Consumer. `Sender` and `SyncSender` implement `Clone` (so you can make multiple producers) but `Receiver` does not.
- `send()` and `recv()` return `Result`. If they return `Err`, it means the counterpart `Sender` or `Receiver` is dropped and the channel is closed.

Unbounded Channels

You get an unbounded and asynchronous channel with `mpsc::channel()`:

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

▼ *Speaker Notes*

This slide should take about 2 minutes.

- An unbounded channel will allocate as much space as is necessary to store pending messages. The `send()` method will not block the calling thread.
- A call to `send()` will abort with an error (that is why it returns `Result`) if the channel is closed. A channel is closed when the receiver is dropped.

Bounded Channels

With bounded (synchronous) channels, `send()` can block the current thread:

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::sync_channel(3);

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

- Calling `send()` will block the current thread until there is space in the channel for the new message. The thread can be blocked indefinitely if there is nobody who reads from the channel.
- Like unbounded channels, a call to `send()` will abort with an error if the channel is closed.
- A bounded channel with a size of zero is called a “rendezvous channel”. Every send will block the current thread until another thread calls `recv()`.

Send and Sync

This segment should take about 15 minutes. It contains:

Slide	Duration
Marker Traits	2 minutes
Send	2 minutes
Sync	2 minutes
Examples	10 minutes

Marker Traits

How does Rust know to forbid shared access across threads? The answer is in two traits:

- `Send` : a type `T` is `Send` if it is safe to move a `T` across a thread boundary.
- `Sync` : a type `T` is `Sync` if it is safe to move a `&T` across a thread boundary.

`Send` and `Sync` are unsafe traits. The compiler will automatically derive them for your types as long as they only contain `Send` and `Sync` types. You can also implement them manually when you know it is valid.

▼ *Speaker Notes*

This slide should take about 2 minutes.

- One can think of these traits as markers that the type has certain thread-safety properties.
- They can be used in the generic constraints as normal traits.

Send

> A type `T` is `Send` if it is safe to move a `T` value to another thread.

The effect of moving ownership to another thread is that *destructors* will run in that thread. So the question is when you can allocate a value in one thread and deallocate it in another.

▼ *Speaker Notes*

This slide should take about 2 minutes.

As an example, a connection to the SQLite library must only be accessed from a single thread.

Sync

> A type `T` is `Sync` if it is safe to access a `T` value from multiple threads at the same time.

More precisely, the definition is:

> `T` is `Sync` if and only if `&T` is `Send`

▼ *Speaker Notes*

This slide should take about 2 minutes.

This statement is essentially a shorthand way of saying that if a type is thread-safe for shared use, it is also thread-safe to pass references of it across threads.

This is because if a type is Sync it means that it can be shared across multiple threads without the risk of data races or other synchronization issues, so it is safe to move it to another thread. A reference to the type is also safe to move to another thread, because the data it references can be accessed from any thread safely.

Examples

`Send + Sync`

Most types you come across are `Send + Sync`:

- `i8`, `f32`, `bool`, `char`, `&str`, ...
- `(T1, T2)`, `[T; N]`, `&[T]`, `struct { x: T }`, ...
- `String`, `Option<T>`, `Vec<T>`, `Box<T>`, ...
- `Arc<T>`: Explicitly thread-safe via atomic reference count.
- `Mutex<T>`: Explicitly thread-safe via internal locking.
- `mpsc::Sender<T>`: As of 1.72.0.
- `AtomicBool`, `AtomicU8`, ...: Uses special atomic instructions.

The generic types are typically `Send + Sync` when the type parameters are `Send + Sync`.

`Send + !Sync`

These types can be moved to other threads, but they're not thread-safe. Typically because of interior mutability:

- `mpsc::Receiver<T>`
- `Cell<T>`
- `RefCell<T>`

`!Send + Sync`

These types are safe to access (via shared references) from multiple threads, but they cannot be moved to another thread:

- `MutexGuard<T: Sync>`: Uses OS level primitives which must be deallocated on the thread which created them. However, an already-locked mutex can have its guarded variable read by any thread with which the guard is shared.

`!Send + !Sync`

These types are not thread-safe and cannot be moved to other threads:

- `Rc<T>`: each `Rc<T>` has a reference to an `RcBox<T>`, which contains a non-atomic reference count.
- `*const T`, `*mut T`: Rust assumes raw pointers may have special concurrency considerations.

Shared State

This segment should take about 30 minutes. It contains:

Slide	Duration
Arc	5 minutes
Mutex	15 minutes
Example	10 minutes

Arc

`Arc<T>` allows shared, read-only ownership via `Arc::clone`:

```
use std::sync::Arc;
use std::thread;

/// A struct that prints which thread drops it.
#[derive(Debug)]
struct WhereDropped(Vec<i32>);

impl Drop for WhereDropped {
    fn drop(&mut self) {
        println!("Dropped by {:?}", thread::current().id())
    }
}

fn main() {
    let v = Arc::new(WhereDropped(vec![10, 20, 30]));
    let mut handles = Vec::new();
    for i in 0..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            // Sleep for 0-500ms.
            std::thread::sleep(std::time::Duration::from_millis(500 - i * 100));
            let thread_id = thread::current().id();
            println!("{thread_id:?}: {v:?}");
        }));
    }

    // Now only the spawned threads will hold clones of `v`.
    drop(v);

    // When the last spawned thread finishes, it will drop `v`'s contents.
    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

▼ *Speaker Notes*

This slide should take about 5 minutes.

- `Arc` stands for "Atomic Reference Counted", a thread safe version of `Rc` that uses atomic operations.
- `Arc<T>` implements `Clone` whether or not `T` does. It implements `Send` and `Sync` if and only if `T` implements them both.
- `Arc::clone()` has the cost of atomic operations that get executed, but after that the use of the `T` is free.
- Beware of reference cycles, `Arc` does not use a garbage collector to detect them.
    - `std::sync::Weak` can help.

Mutex

`Mutex<T>` ensures mutual exclusion *and* allows mutable access to `T` behind a read-only interface (another form of interior mutability):

```
use std::sync::Mutex;

fn main() {
    let v = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());

    {
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }

    println!("v: {:?}", v.lock().unwrap());
}
```

Notice how we have a `impl<T: Send> Sync for Mutex<T>` blanket implementation.

▼ *Speaker Notes*

This slide should take about 14 minutes.

- `Mutex` in Rust looks like a collection with just one element — the protected data.
  - It is not possible to forget to acquire the mutex before accessing the protected data.
- You can get an `&mut T` from an `&Mutex<T>` by taking the lock. The `MutexGuard` ensures that the `&mut T` doesn't outlive the lock being held.
- `Mutex<T>` implements both `Send` and `Sync` if and only if `T` implements `Send`.
- A read-write lock counterpart: `RwLock`.
- Why does `lock()` return a `Result`?
  - If the thread that held the `Mutex` panicked, the `Mutex` becomes "poisoned" to signal that the data it protected might be in an inconsistent state. Calling `lock()` on a poisoned mutex fails with a `PoisonError`. You can call `into_inner()` on the error to recover the data regardless.

Example

Let us see `Arc` and `Mutex` in action:

```
use std::thread;
// use std::sync::{Arc, Mutex};

fn main() {
    let v = vec![10, 20, 30];
    let mut handles = Vec::new();
    for i in 0..5 {
        handles.push(thread::spawn(|| {
            v.push(10 * i);
            println!("v: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

▼ *Speaker Notes*

This slide should take about 8 minutes.

Possible solution:

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let v = Arc::new(Mutex::new(vec![10, 20, 30]));
    let mut handles = Vec::new();
    for i in 0..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            let mut v = v.lock().unwrap();
            v.push(10 * i);
            println!("v: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

Notable parts:

- `v` is wrapped in both `Arc` and `Mutex`, because their concerns are orthogonal.
  - Wrapping a `Mutex` in an `Arc` is a common pattern to share mutable state between threads.
- `v: Arc<_>` needs to be cloned to make a new reference for each new spawned thread. Note `move` was added to the lambda signature.
- Blocks are introduced to narrow the scope of the `LockGuard` as much as possible.

Exercises

This segment should take about 1 hour and 10 minutes. It contains:

Slide	Duration
Dining Philosophers	20 minutes
Multi-threaded Link Checker	20 minutes
Solutions	30 minutes

Dining Philosophers

The dining philosophers problem is a classic problem in concurrency:

> Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a chopstick between each plate. The dish served is spaghetti which requires two chopsticks to eat. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right chopstick. Thus two chopsticks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both chopsticks.

You will need a local Cargo installation for this exercise. Copy the code below to a file called `src/main.rs`, fill out the blanks, and test that `cargo run` does not deadlock:

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
use std::time::Duration;

struct Chopstick;

struct Philosopher {
    name: String,
    // left_chopstick: ...
    // right_chopstick: ...
    // thoughts: ...
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        // Pick up chopsticks...
        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Hypatia", "Plato", "Aristotle", "Pythagoras"];

fn main() {
    // Create chopsticks

    // Create philosophers

    // Make each of them think and eat 100 times

    // Output their thoughts
}
```

You can use the following `Cargo.toml`:

```
[package]
name = "dining-philosophers"
version = "0.1.0"
edition = "2024"
```

▼ *Speaker Notes*

This slide should take about 20 minutes.

- Encourage students to focus first on implementing a solution that “mostly” works.
- The deadlock in the simplest solution is a general concurrency problem and highlights that Rust does not automatically prevent this sort of bug.

Multi-threaded Link Checker

Let us use our new knowledge to create a multi-threaded link checker. It should start at a webpage and check that links on the page are valid. It should recursively check other pages on the same domain and keep doing this until all pages have been validated.

For this, you will need an HTTP client such as `reqwest`. You will also need a way to find links, we can use `scraper`. Finally, we'll need some way of handling errors, we will use `thiserror`.

Create a new Cargo project and `reqwest` it as a dependency with:

```
cargo new link-checker
cd link-checker
cargo add --features blocking,rustls-tls reqwest
cargo add scraper
cargo add thiserror
```

> If `cargo add` fails with `error: no such subcommand`, then please edit the `Cargo.toml` file by hand. Add the dependencies listed below.

The `cargo add` calls will update the `Cargo.toml` file to look like this:

```
[package]
name = "link-checker"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

You can now download the start page. Try with a small site such as `https://www.google.org/`.

Your `src/main.rs` file should look something like this:

```
use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}: {err}");
            }
        }
    }
    Ok(link_urls)
}

fn main() {
    let client = Client::new();
    let start_url = Url::parse("https://www.google.org").unwrap();
    let crawl_command = CrawlCommand{ url: start_url, extract_links: true };
    match visit_page(&client, &crawl_command) {
        Ok(links) => println!("Links: {links:#?}"),
        Err(err) => println!("Could not extract links: {err:#}"),
    }
}
```

Run the code in `src/main.rs` with

```
cargo run
```

Tasks

- Use threads to check the links in parallel: send the URLs to be checked to a channel and let a few threads check the URLs in parallel.
- Extend this to recursively extract links from all pages on the `www.google.org` domain. Put an upper limit of 100 pages or so so that you don't end up being blocked by the site.

▼ *Speaker Notes*

This slide should take about 20 minutes.

- This is a complex exercise and intended to give students an opportunity to work on a larger project than others. A success condition for this exercise is to get stuck on some "real" issue and work through it with the support of other students or the instructor.

Solutions

Dining Philosophers

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
use std::time::Duration;

struct Chopstick;

struct Philosopher {
    name: String,
    left_chopstick: Arc<Mutex<Chopstick>>,
    right_chopstick: Arc<Mutex<Chopstick>>,
    thoughts: mpsc::SyncSender<String>,
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        println!("{} is trying to eat", &self.name);
        let _left = self.left_chopstick.lock().unwrap();
        let _right = self.right_chopstick.lock().unwrap();

        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Hypatia", "Plato", "Aristotle", "Pythagoras"];

fn main() {
    let (tx, rx) = mpsc::sync_channel(10);

    let chopsticks = PHILOSOPHERS
        .iter()
        .map(|_| Arc::new(Mutex::new(Chopstick)))
        .collect::<Vec<_>>();

    for i in 0..chopsticks.len() {
        let tx = tx.clone();
        let mut left_chopstick = Arc::clone(&chopsticks[i]);
        let mut right_chopstick =
            Arc::clone(&chopsticks[(i + 1) % chopsticks.len()]);

        // To avoid a deadlock, we have to break the symmetry
        // somewhere. This will swap the chopsticks without deinitializing
        // either of them.
        if i == chopsticks.len() - 1 {
            std::mem::swap(&mut left_chopstick, &mut right_chopstick);
        }

        let philosopher = Philosopher {
            name: PHILOSOPHERS[i].to_string(),
            thoughts: tx,
            left_chopstick,
            right_chopstick,
        };
```

```
                philosopher.eat();
                philosopher.think();
            }
        });
    }

    drop(tx);
    for thought in rx {
        println!("{thought}");
    }
}
```

Link Checker

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;

use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}: {err}");
            }
        }
    }
    Ok(link_urls)
}

struct CrawlState {
    domain: String,
    visited_pages: std::collections::HashSet<String>,
}

impl CrawlState {
    fn new(start_url: &Url) -> CrawlState {
        let mut visited_pages = std::collections::HashSet::new();
        visited_pages.insert(start_url.as_str().to_string());
        CrawlState { domain: start_url.domain().unwrap().to_string(), visited_pages }
    }
```

```
        let Some(url_domain) = url.domain() else {
            return false;
        };
        url_domain == self.domain
    }

    /// Mark the given page as visited, returning false if it had already
    /// been visited.
    fn mark_visited(&mut self, url: &Url) -> bool {
        self.visited_pages.insert(url.as_str().to_string())
    }
}

type CrawlResult = Result<Vec<Url>, (Url, Error)>;
fn spawn_crawler_threads(
    command_receiver: mpsc::Receiver<CrawlCommand>,
    result_sender: mpsc::Sender<CrawlResult>,
    thread_count: u32,
) {
    // To multiplex the non-cloneable Receiver, wrap it in Arc<Mutex<_>>.
    let command_receiver = Arc::new(Mutex::new(command_receiver));

    for _ in 0..thread_count {
        let result_sender = result_sender.clone();
        let command_receiver = Arc::clone(&command_receiver);
        thread::spawn(move || {
            let client = Client::new();
            loop {
                let command_result = {
                    let receiver_guard = command_receiver.lock().unwrap();
                    receiver_guard.recv()
                };
                let Ok(crawl_command) = command_result else {
                    // The sender got dropped. No more commands coming in.
                    break;
                };
                let crawl_result = match visit_page(&client, &crawl_command) {
                    Ok(link_urls) => Ok(link_urls),
                    Err(error) => Err((crawl_command.url, error)),
                };
                result_sender.send(crawl_result).unwrap();
            }
        });
    }
}

fn control_crawl(
    start_url: Url,
    command_sender: mpsc::Sender<CrawlCommand>,
    result_receiver: mpsc::Receiver<CrawlResult>,
) -> Vec<Url> {
    let mut crawl_state = CrawlState::new(&start_url);
    let start_command = CrawlCommand { url: start_url, extract_links: true };
    command_sender.send(start_command).unwrap();
    let mut pending_urls = 1;

    let mut bad_urls = Vec::new();
    while pending_urls > 0 {
        let crawl_result = result_receiver.recv().unwrap();
        pending_urls -= 1;

        match crawl_result {
            Ok(link_urls) => {
                for url in link_urls {
                    if crawl_state.mark_visited(&url) {
                        let extract_links = crawl_state.should_extract_links(&url);
                        let crawl_command = CrawlCommand { url, extract_links };
                        command_sender.send(crawl_command).unwrap();
                        pending_urls += 1;
                    }
```

```
            Err((url, error)) => {
                bad_urls.push(url);
                println!("Got crawling error: {:#}", error);
                continue;
            }
        }
    }
    bad_urls
}

fn check_links(start_url: Url) -> Vec<Url> {
    let (result_sender, result_receiver) = mpsc::channel::<CrawlResult>();
    let (command_sender, command_receiver) = mpsc::channel::<CrawlCommand>();
    spawn_crawler_threads(command_receiver, result_sender, 16);
    control_crawl(start_url, command_sender, result_receiver)
}

fn main() {
    let start_url = reqwest::Url::parse("https://www.google.org").unwrap();
    let bad_urls = check_links(start_url);
    println!("Bad URLs: {:#?}", bad_urls);
}
```

Welcome

“Async” is a concurrency model where multiple tasks are executed concurrently by executing each task until it would block, then switching to another task that is ready to make progress. The model allows running a larger number of tasks on a limited number of threads. This is because the per-task overhead is typically very low and operating systems provide primitives for efficiently identifying I/O that is able to proceed.

Rust’s asynchronous operation is based on “futures”, which represent work that may be completed in the future. Futures are “polled” until they signal that they are complete.

Futures are polled by an async runtime, and several different runtimes are available.

Comparisons

- Python has a similar model in its `asyncio`. However, its `Future` type is callback-based, and not polled. Async Python programs require a “loop”, similar to a runtime in Rust.
- JavaScript’s `Promise` is similar, but again callback-based. The language runtime implements the event loop, so many of the details of Promise resolution are hidden.

Schedule

Including 10 minute breaks, this session should take about 3 hours and 30 minutes. It contains:

Segment	Duration
Async Basics	40 minutes
Channels and Control Flow	20 minutes
Pitfalls	55 minutes
Exercises	1 hour and 10 minutes