Concurrency and Tasks in Rust 0.10

Processes vs. Threads

Both threads and processes are abstractions the operating system provides. Through the process abstraction, we can provide each program with the illusion that it owns the machine it is running on. This means giving each program its own address space, stack, registers and program counter, in addition to CPU time. Switching from a running process to a waiting process then requires the processor to make a context switch. This context switch involves saving the current thread's state to a process control block and restoring the state of the sleeping process. When making the context switch from one process to another we also have to switch address spaces, which is a fairly expensive process.

A thread is a bit smaller than a process, and is often defined as the smallest possible unit of schedulable computation. Each thread retains the ability to execute a sequence of instructions by having separate program counters, registers and stack, threads share an address space. This means a context switch between threads does not require switching the address space, and can be done with lower cost.

Cost of Creation

Since processes have their own address space, whenever a new process is created, this address space must also be created. Threads are therefore lighter to create, as they attach to existing address space instead of having to allocate new space in memory.

Shared Memory

Since threads share address space it is relatively easy for programmers to have them communicate. Processes are more difficult, and either require the use of special interfaces like pipes and message passing or very careful manual memory management.

Tasks in Rust

Rust tasks are a special type of thread. While tasks share memory space, only immutable memory can be shared. When spawning a task you provide a closure to pass to the function. By only accepting owned closures, Rust can limit data shared between tasks to immutable data, since the parent task owns the data and allows child tasks to reference it. As soon as a child task tries to modify that owned box, that child task would have to take ownership of the box, violating the closure. Beyond closures, any type that is of the `Send` kind can be passed between tasks. `Send` types can only contain owned types, and thus do not violate ownership rules, but do not need to be explicitly stated as closures.

Task Implementation

Rust implements tasks in two ways, based on which crate you use. The default native scheduling crate allows you to create tasks that use one-to-one scheduling. This means each task is mapped to a single operating system (OS) kernel provided threads. Stated simply, for every task created by the user, there is a separate OS thread in which the task is run. The green scheduling crate provides the ability to create green threads. Green threads are user space threads that map to kernel threads in an M:N manner. Green threads take a bit more effort to set up in Rust and while they can be faster to spawn, they have poorer I/O performance and slower context switches. This chapter focuses on the first.

Example 1: Task Creation

```
use std::task;

fn printStr(s: &str) {
    println!( "{:s}", s );
}

fn main() {
    task::spawn( proc(){ printStr("A"); });
    task::spawn( proc(){ printStr("B"); });
}
```

In Example 1, a simple task is created. Since we did not specify which task library to use, these tasks will be created using the native thread model, with each task having its corresponding kernel thread. In this simple example there is no need for shared data.

Shared Memory Space

As stated before, the shared memory space of Rust tasks is immutable. However, Rust does allow programmers to disregard the built-in safety mechanisms and access memory unsafely as show in Example 2.

Example 2: Failed Data Sharing "‘rust use std::task;

static mut count: int = 0;

fn main() {

```
for _ in range(0, 10000) {
    task::spawn( proc(){
        for _ in range(0, 100) {
            unsafe {
                count += 1;
            }
        }
    });
}

println!("Result should be 1,000,000");
unsafe {
    println!("main: done with both count = {}", count);
}

}
```

Here, 10,000 threads are created and each increments a global mutable variable 100 times. Even though the simple increment operationcount += 1;“‘ seems like it should be atomic is not guaranteed to be, and since we do not control the scheduling of each task any task can access count and perform this increment at anytime. This will often lead to incorrect results. Beyond incorrect results, the results are inconsistent, making this a very difficult bug to identify and fix in a more complex program.

The solution to our problem is to safely share this data. Rust provides several higher-level data structures that control access to memory. Example 3 shows two of these, the Arc and RWLock structures.

Example 3: Safe Data Sharing

```
extern crate sync;

use sync::{RWLock, Arc};

fn main() {

    let lock1 = Arc::new(RWLock::new(0));

    for _ in range(0, 10000) {

        let lock2 = lock1.clone();

        spawn(proc() {

            let mut count = lock2.write();
            println!("before -  count = {}", *count);
            for _ in range (0, 100) {
                *count += 1;
            }

            let count = count.downgrade();

            println!("after -  count = {}", *count);
        });
    }

    println!("Result should be 1000000");
    println!("Counter is: {}", *lock1.read());
}
```

In this example, an Arc is used to share data between tasks. An acronym for Atomically Reference Counted data, the Arc creates a pointer to share data between tasks. Since it is owned by the start task, and each task gets a unique cloned version, it is immutable and can be safely passed into owned closures. The Arc only provides a pointer to share a memory location between tasks, and doesn't guarantee any form of safe or controlled access though. To control access to the data here, we must use a RWLock or similar type. The RWLock allows multiple tasks to conduct immutable actions, like reading data, with that portion of memory. When a task needs to mutate the RWLock data it must request write privileges. When these are granted, all other tasks trying to access the data will block until the writing task is finished and has relinquished control.

Of course, sometimes we want to look at the underlying structures used for this. Example 4 shows how we can safely share data between tasks using senders and receivers.

Example 4: Senders, Receivers and a Channel “‘rust use std::task; use std::comm::channel;

fn main() { let mut count: int = 0;

```
for _ in range(0, 10000) {
    let (tx, rx) = channel();
    let count2 = count;

    task::spawn( proc(){
        let mut mut_count = count2;

    for _ in range (0,100) {
            mut_count += 1;
        }
        tx.send(mut_count);
    });
    count = rx.recv();
}

println!("Result should be 1000000");
println!("main: done with both count = {}", count);
```

} Similar to the previous example, we need a way to pass data from the parent task to the child tasks and then in between each child task. The most basic way Rust does this is through the creation of achannel(), with aSendertype on one end and aReceivertype on the other end. In this example we pass data from the parent task to child tasks through an immutable variable, soSendersandReceiversare only needed to communicate from the child threads back to the parent thread. Within theforloop to spawn a new thread we declare a Sender,txand a Receiverrx"‘, that allow the soon to be spawned child task to send data back to its parent.

Try it!

Example 5: Deadlock "‘rust extern crate sync;

use sync::{Mutex, Arc};

fn increment(count: int) -> int{ return count + 1;
}

fn main() {

```
let mutex1 = Arc::new(Mutex::new(0));
let mutex2 = mutex1.clone();

spawn(proc() {

    let mut val = mutex2.lock();
    *val = increment(*val);


});

let value = mutex1.lock();
while *value != 2 {
    value.cond.wait();
}
```

```
println!("Lock was never released in spawned proc(), so I will never be reached!");
```

} ‘‘‘ 1. Explain why this code deadlocks. 2. Use different features of Rust to write your own program that deadlocks.

References

Arpaci-Dusseau, Andrea and Remzi. Operating Systems: Three Easy Pieces. http://pages.cs.wisc.edu/~remzi/OSTEP/

Evans, David. University of Virginia, CS 4414: Operating Systems. Spring 2014

Rust Docs

http://static.rust-lang.org/doc/master/guide-tasks.html http://static.rust-lang.org/doc/master/green/index.html http://static.rust-lang.org/doc/master/native/index.html http://static.rust-lang.org/doc/master/std/task/index.html