Synchronous Programming with User-Level Threads in C∀

November 4, 2019

Presented by: Thierry Delisle

UNIVERSITY OF WATERLOO

Intro

- Synchronous Programming
- Synchronous Programming in C∀(C-for-all)
  - Threads
  - Monitors
- Using Synchronous Programming with other paradigms

Synchronous Programming

Synchronous vs Asynchronous

- Functions which can't immediately return a result will :
  a. Block the caller until completed (synchronous).
  b. Return immediately and communicate the result later through an alternate method (asynchronous).

- Synchronous programming tools: function calls
- Asynchronous programming tools: callbacks, events, Futures/Promises, "async/await"

Sync vs Async example: naïve stddev

Given 2 parallel programming APIs, what does a standard deviation calculation look like?

```
long stddev(long[] data);
```

Synchronous API:

```
// Sum of an array
long par_sum(long[] data);

// Element wise subtract scalar
long[] par_sub(long[] data, long s);

// Element-wise power by scalar
long[] par_pow(long[] base, long s);
```

Asynchronous API:

```
// Sum of an array
void par_sum(long[] data, lambda l);

// Element wise subtract scalar
void par_sub(long[] data, long s, lambda l);

// Element-wise power by scalar
void par_pow(long[] base, long s, lambda l);
```

Sync vs Async example: naïve stddev

Synchronous

```
long stddev(long[] data) {
    long count = data.size();

    long mean = par_sum(data) / count;

    long[] diff = par_sub(data, mean);
    long[] diffSqr = par_pow(diff, 2);

    long variance = par_sum(diffSqr);

    return sqrt(variance / count);
}
```

Sync vs Async example: naïve stddev

Synchronous

```
long stddev(long[] data) {
    long count = data.size();

    long mean = par_sum(data) / count;

    long[] diff = par_sub(data, mean);
    long[] diffSqr = par_pow(diff, 2);

    long variance = par_sum(diffSqr);

    return sqrt(variance / count);
}
```

Asynchronous

```
long stddev(long[] data) {
    long count = data.size();
    par_sum(data, (s) {
        long mean = s / count;
        par_sub(data, mean, (d) {
            par_pow(d, 2, (ds) {
                par_sum(ds, (v) {
                    long stddev =
                    sqrt( v / count );
                    // Now what?
                }
            })
        })
    })
    return ???;
}
```

Better Async stddev

- Promises
- Async & Await

\+ Looks better

\- Still more code

\- Still more knowledge needed

\- Parent still needs to change

```
async long stddev(long[] data) {
    long count = data.size();

    long mean = await par_sum(data) / count;

    long[] diff = await par_sub(data, mean);
    long[] diffSqr = await par_pow(diff, 2);

    long variance = await par_sum(diffSqr);

    return sqrt(variance / count);
}
```

Synchronous programming

- Better abstraction: Caller sees a regular function
- Simpler code
- Better debugging
  - Stack traces are meaning full
  - Things happen in program order

- But : concurrency and parallelism require *context switching*.
  - Must be able to *pause* a function, do something else and then come back.

Synchronous Programming in C∀

What is C∀ ?

- C∀ is a new programming language created at the University of Waterloo
  - polymorphic
  - non-object-oriented
  - concurrent
  - backwards compatible with C

- C∀ has similar goals to GO, Rust and Scala

https://cforall.uwaterloo.ca

Threads in C∀

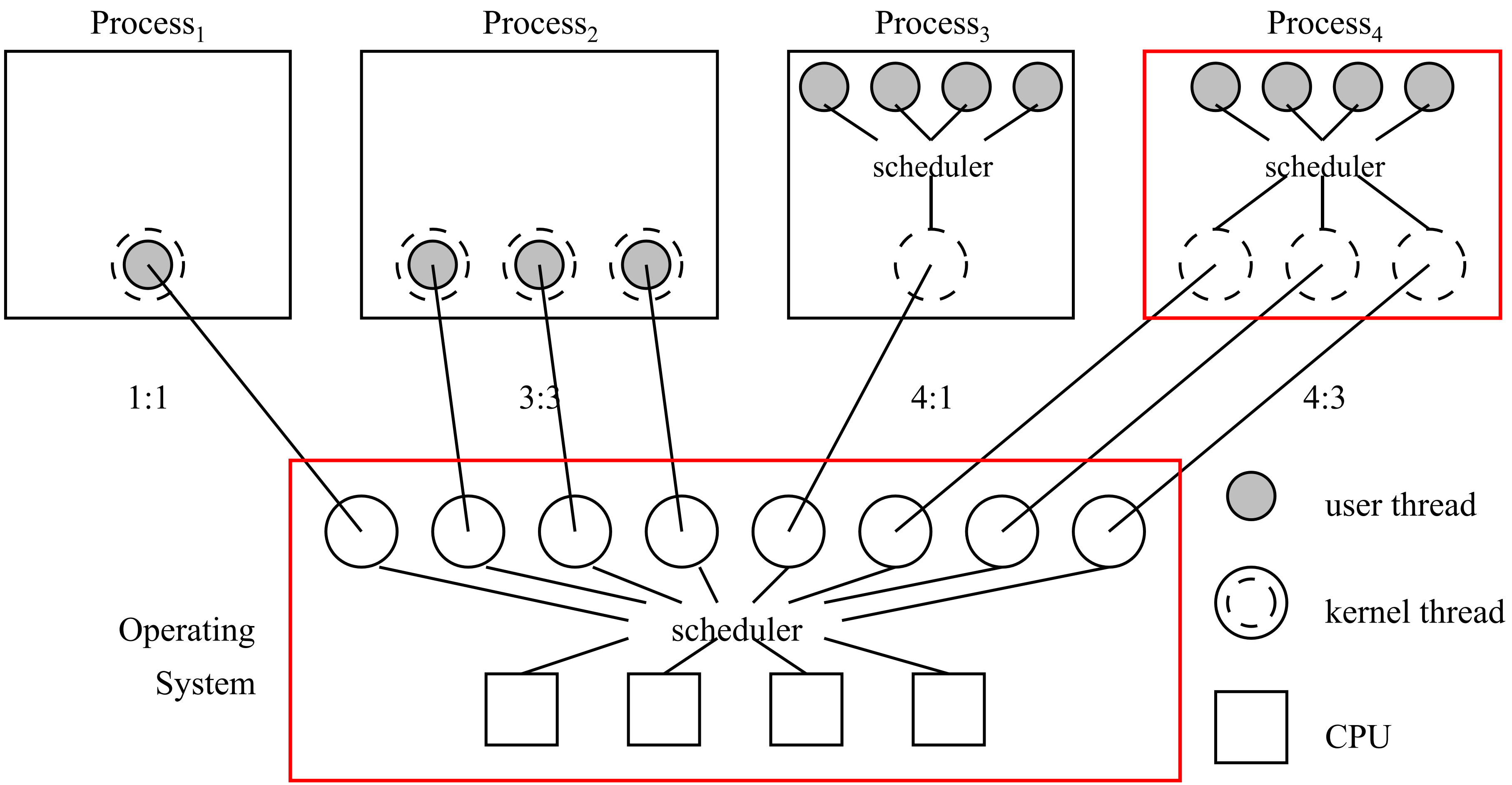
- Concurrent programming in C∀ uses `threads`
- Threads have both a type and a `main` routine
- Construction (`?{}`) starts the thread
- Destruction waits for the thread to finish (*i.e.* join)

```
thread MyThrd {
        size_t id;
};

// Constructor
void ?{}(MyThrd & this, size_t id )
{ t.id = id; }

// Thread main
void main( T & this ) {
        // thread starts here
        stdout | id;
}

// Declaration and construction
MyThread myInstance = { 22 };
```

Threading in C∀

- Threading is based on user-level threading (A.K.A. lightweight threads, green thread, fibers, goroutines, *coroutines*).
- M:N threading model
  - M kernel threads run N user threads.
  - Mapping from user thread to kernel thread is done in user space.
  - Mapping from kernel threads to CPU/hardware thread is done in kernel space as usual
- Benefits:
  - Faster creation and context-switching mean more threads can be created
  - Cheaper threads lead to simpler programming

User-Level threading



Synchronous Programming with User-Level threads

- Threaded matrix sum example
  - Program creates *rows* threads
  - Each thread sums a row
  - Waits for them to finish
  - Add subtotals
  - Profit!

- processor objects specify parallelism (*e.g.* GO_MAX_PROCS)

* *New* gets type from left side of assignment

```
thread Adder {
    int * row, cols, &subtotal; // communication
};
void ?{}(Adder& adder, /*…*/) { /*…*/ }

void main( Adder & adder ) with( adder ){
    // thread starts here
    subtotal = 0;
    for ( c; cols ) { subtotal += row[c]; }
}

int main() {
    const int rows = 10, cols = 1000;
    int matrix[rows][cols], totals[rows], total = 0;
    // initialize matrix
    processor p[3]; // add kernel thread

    Adder * adders[rows];
    for ( r; rows ) { // start threads to sum rows
        adders[r] = new( matrix[r], cols, totals[r] );
    }
    for ( r; rows ) { // wait for threads to finish
        delete( adders[r] ); total += subtotals[r];
    }
    stdout | total;
}
```

Higher-Level Locks

- User space threads require user space locks

- Locks in user-space don’t need support kernel support
  - Language specific locks possible
  - Higher-level locks are possible
  - Custom locks are possible

Higher-Level Locks in C∀

- monitors tie together
  - data,
  - Interface
  - mutual exclusion
- mutexed arguments handle mutual exclusion automatically
- Support simultaneous acquiring multiple monitor
  - Called bulk acquiring

```
monitor BankAccount { int balance; };

void deposit( BankAccount & mutex b,
    int deposit ) {
    b.balance += deposit;
}

void transfer( BankAccount & mutex my,
    BankAccount & mutex your, int me2you
) {
    deposit( my, -me2you ); // debit
    deposit( your, me2you ); // credit
}
```

Higher-Level Locks in C∀ : waitfor

- Monitors support more powerful semantics like `waitfor`
- Waits for function with given monitor to be called
  - Supports conditions
  - Prevents barging
  - Much More

```
monitor BankAccount {…}

void deposit( BankAccount & mutex,
    int );

int withdraw_some(
    BankAccount & mutex b
) {
    when(0 == b.balance)
        waitfor(debosit, b);
    int amount = b.balance;
    b.balance = 0;
    return amount;
}
```

Higher-Level Locks in C∀ : bulks

- `waitfor` works as expected when using
  - Multiple monitors
  - Bulk acquiring
  - Recursion
  - etc.
- Example: Ensure approval arrives before transfer

```
monitor BankAccount {…}

void transfer( BankAccount & mutex my,
    BankAccount & mutex your, int );

void approve( BankAccount & mutex,
    BankAccount & mutex );

void validate(
    BankAccount & mutex my,
    BankAccount & mutex your
) {
    waitfor(approve, my, your );
    waitfor(transfer, my, your);
}
```

Using Cforall with existing APIs

Upgrading async functions to user-threads

- Example: sum large array
- Existing code returns immediately and uses callback on completion

```
// Original
void par_sum(long[] data, lambda l);

// Target
long sum (long[] data);
```

Upgrading async functions to user-threads

- Example: sum large array
- Existing code returns immediately and uses callback on completion

```
// Original
void par_sum(long[] data, lambda l);

// Target
long sum (long[] data);
```

```
long sum (long[] data) {
      // setup required data
      long result;
      semaphore sem = { 0 };

      // call async function
      par_sum(data, (int sum) {
            result = sum;
            V(sem);
      });

      // wait for result
      P(sem);

      // return
      return result;
}
```

Upgrading blocking functions to user-threads

- Example: sum large array
- Existing code blocks kernel thread for duration of the function

```
// Original (blocks kernel thread)
long par_sum(long[] data);

// Target (blocks user thread)
long sum (long[] data);
```

Upgrading blocking functions to user-threads

- Example: sum large array
- Existing code blocks kernel thread for duration of the function

```
// Original (blocks kernel thread)
long par_sum(long[] data);

// Target (blocks user thread)
long sum (long[] data);
```

```
struct request {
        long[] data; long result;
        semaphore sem = { 0 };
};

void * callit(request * req) {
        req->result = par_sum(req->inputs);
        V(req->sem);
}

long sum (long[] data) {
    request req = { inputs }; // setup required data

    pthread_t thrd; // create thread to handle call
    int err = pthread_create(&thrd, 0p, callit, &req);
    // handle err …

    P(req.sem);

    err = pthread_join(&thrd, 0p); // handle err …
    return req.result;
}
```

Conclusion

- Synchronous programming can lead to simpler programs
- Multiple paradigms can mix
- Check out C∀

QUESTIONS