Concurrency

TTK4145 2022 02

Blinkenlights

Blink 1 LED -> Trivial

Blink 2 LEDs -> Hard

- Option 1: Interleave the sleeps
- Option 2: Short sleep with counter
- Option 3: Interrupts? But we have a (very) limited number
  - (Because they are a hardware mechanism)

Desire: Two separate functions

- *Real* desire: Interleaved execution, without interleaved code

Key realization:

- Functions don't do anything!

Blinkenthreads

Functions only do something
if they are being run

We can run multiple functions "at the same time"
by swapping between them very fast

This is beneficial if the tasks they perform are independent

Tasks are rarely *entirely* independent

Swapping tasks can have provoke nasty behaviors

Some definitions

Thread:
the thing that holds all the data for a function

Scheduler:
the thing that decides what thread to run next

Context switch:
the act of swapping

Race condition:
when the result depends on ordering in time

Synchronization:
ways to force things to run in some order (for correctness)

First principles

Intelligent design

"This is how it has to be" vs
"This is how it is"

It's not magic

- We can understand how it works
- By thinking carefully (and for a long time), you could do it yourself

It's not luck (usually)

- There is often only one way to do it
- When there are more ways, there is usually one that is best
- When there isn't one that is best, it's usually the first one that wins
- Etc

Threads

From scratch

How do variables work?

- A variable holds data
- The data has to "live" somewhere
  - It needs memory allocated to it
- The data must be referenced by something (must be used)
  - Fixed location: Global variables, heap-allocated memory
  - Relative location: Stack-allocated memory

How do constants work?

- Constants are read-only
- More ways to reference constants
  - Like a regular variable, but with compile-time check for constant-ness
  - Encoded directly in the machine code instructions
  - Loaded into (OS- or MMU-enforced) read-only memory

How do functions work?

Functions are "blocks" of code

- A collection of CPU instructions
- Some kind of "call" and "return" instructions

Can be called from several places

- Either: Local variables must have relative references (hence stack-allocation)
- Or: ALL variables must have fixed location AND no recursion

Requirements:

- A memory region (for this fn's local variables, and any enclosing fn's variables)
- A stack pointer (where the latest local variable is)
- A frame pointer (optional, where the stack pointer started when fn was called)
- Instructions use SP (or FP) with offsets for memory references

Thread swapping & starting

Swapping a thread

- Call a "yield" function
- Save context to stack: register values & stack pointer
- Run some "reschedule" function to find a new thread
- Restore (different) stack pointer and registers
- Return (this will "return" into a different place than we "call"ed from!)

Starting a thread

- Allocate a new stack (probably on the heap!)
- Put initial stack pointer and registers on the fresh stack
- Register the thread data structure with the scheduler
- Scheduler will "restore" the initial data when it re-schedules

Swapping threads

```
static void* ctxSwitchStack;

__attribute__((naked)) void sched_yield(void){
    SAVE_CONTEXT();
    MOVE_SP(ctxSwitchStack);

    reschedule();

    RESTORE_CONTEXT();
    asm volatile ( "ret" );
}
```

Making a thread

```
static const uint16_t contextSize = 40;
Thread os_spawn(void fn(void), uint16_t stackSize){
    Thread t = osthread_new(stackSize + contextSize);
    t->owner = os_thisThread();

    // Put stuff on the stack, so we can pop it the first time
    // the scheduler context-switches to the new thread
    t->stack.ptr = osthread_initializeStack(t->stack.ptr, fn);
    atom sched_state_setRunnable(t);

    return t;
}
```

Initializing stack

```
uint8_t* osthread_initializeStack(
    uint8_t* stackPtr, void fn(void)
){
    // Shove entry point on stack
    uint16_t fnAddr = (uint16_t)fn;
    *stackPtr-- = (uint8_t)(fnAddr      & (uint16_t)0x00ff);
    *stackPtr-- = (uint8_t)(fnAddr>>8   & (uint16_t)0x00ff);

    // Initial values for registers, as if saved by SAVE_CONTEXT
    *stackPtr-- = 0x00; // R0
    *stackPtr-- = 0x80; // SREG: Interrupts Enabled

    // R1..31
    for(uint8_t r = 1; r <= 31; r++){
        *stackPtr-- = 0x00;
    }
    return stackPtr;
}
```

When do we swap?

Cooperative scheduling:

Calls to yield (reschedule) are inserted at key locations in code

- Either by the programmer (manually), or by a compiler (automatically)

Preemptive scheduling:

Calls to yield are triggered by an external time source

- A timer interrupt service routine
- This automatically (in hardware) moves the Program Counter
- Usually a fixed interval, often 1ms

How does sleep work?

Two main ways:

- Count cycles (usually called "delay")
  (No OS required, but cannot do other stuff)
- Register a "wake time" and yield
  (OS required, but can do other stuff)

Sleeping is "waiting" for time

- Calling sleep tells the scheduler to make the thread "not runnable"
- Rescheduling function checks the time,
  "wakes" thread by making it runnable again

Rescheduling

```
static void reschedule(void){
    checkStackOverflow((OSThread*)currentThread);
    wakeSleepingThreads();

    uint8_t numRunnable = cdlist_length(&list_runnable);

    if(numRunnable != 0){
        uint8_t r = rand_r(&threadSelectRandCtx) % numRunnable;
        // Get thread-pointer via offset from its schedList-pointer
        currentThread =
          enclosing(cdlist_idx(&list_runnable, r), OSThread, schedList);
    } else {
        currentThread = idleThread;
    }
}
```

Sleeping and waking

```
void os_sleepUntil(Time time){
    atom {
        os_thisThread()->wakeTime = time;
        sched_state_setSleeping(os_thisThread());
    }
    sched_yield();
}

static void wakeSleepingThreads(void){
    Time now = os_time_now();
    cdlist_foreach(item, list_sleeping){
        OSThread* t = enclosing(item, OSThread, schedList);
        if(os_time_cmp(now, t->wakeTime) >= 0){
            cdlist_unlink(item);
            sched_state_setRunnable(t);
        }
    }
}
```

How does sync work?

- A resource should only be taken by one thread at a time
  - Taking a resource should make others unable to take it
  - Trying to take a resource that is in use should block
  - Same idea as sleeping, but a different "waiting-list"
- The resource is the waiting-list
  - The resource data structure holds the list of those waiting for it
    - (Otherwise, we would have to scan all threads -> inefficient)
  - Releasing a resource makes a waiting thread runnable

Acquiring

```
void os_resource_acquire(Resource* r){
    OSThread* t = os_thisThread();

    // Cannot os_yield inside an atom:
    //  check performed inside atom, os_yield performed outside.
    //  double yield (in case of yield from tick) has no ill effect.
    uint8_t resourceAvailable = 0;
    atom {
        cdlist_append(&r->handle, &t->blockList);
        resourceAvailable = (cdlist_idx(&r->handle, 0) == &t->blockList);
        if(!resourceAvailable){
            sched_state_setBlockedResource(t);
        }
    }

    if(!resourceAvailable){
        sched_yield();
        // Thread should only become runnable once we have the resource
    }
}
```

Releasing

```
void os_resource_release(Resource* r){
    OSThread* t = sched_currentThread();

    atom {
        cdlist_unlink(&t->blockList);
        cdlist_reset(&t->blockList);
        if(cdlist_length(&r->handle) > 0){
            OSThread* nextInLine =
                enclosing(cdlist_idx(&r->handle, 0), OSThread, blockList);
            sched_state_setRunnable(nextInLine);
        }
    }
}
```

How do OS calls work

- We don't want programs to access OS internal memory (or other programs)
  - Hardware mechanism: syscall
    - Moves program counter and performs privilege elevation (sort of like a "software interrupt")
  - Hardware mechanism: Memory management unit
    - Transforms memory addresses from virtual to physical
- Want environment access to be uniform
  - printf = write to file #0, but writes to different windows
  - Process: collection of threads + environment (stdio, cmdline args, etc)

So far

- Threads hold the data to run functions
- Resources hold data to block threads
  - And time is a special kind of resource
- The scheduler selects among runnable threads
- Preemptive scheduling can swap at "any" time
  - As seen from the thread - it can be interrupted at any time
  - This can cause unpredictable behavior when sharing data
- Cooperative scheduling requires manual work
  - Somewhat defeats the purpose of making several threads

AVR OS on GitHub:

https://github.com/klasbo/avr-os/

Blocked externally

How to wait for outside events efficiently

Slow external hardware

Computers only do useful things
if they interact with the outside world

The outside world is a lot slower than the computer

- This is the foundation for concurrency:
  Dealing with several things at the same time (as oppose to *doing*)

We don't want threads to wait for hardware

Solution: Two modes of operation

- Nonblocking: Returns immediately, but not necessarily with data
- Blocking: Returns when there is data

Blocking: good actually

- Blocking calls are easier to deal with
  - Reading from a file actually returns its contents, etc
- Blocking calls require the use of more threads
  - When the thread is waiting, run something else
- However...
- OS thread context switching has costs
  - The time to swap threads can be long
  - The time before a thread swap occurs can be long
  - No guarantee that we swap to thread in the same program
  - => Most significant in lots-of-small-actions low-latency applications (servers)

Nonblocking: also good

- Solution:
  - Use fewer OS threads
  - Use nonblocking calls
    - (epoll, select, WaitForMultipleObjects, IO Completion Ports, etc)
  - Make another layer of task swapping infrastructure ourselves
- New problem:
  - Preemptive scheduling requires HW mechanism to swap tasks
    - Not available in a software-only layer, must do cooperative scheduling
  - Can't just call the function for a different task as part of a yield
    - "Yield inside yield" can cause uncontrolled stack depth

Callbacks

The easiest solution

Runtime: while(wait for event){ process event }

Our code: Tell the runtime what "process" means

-> Pass a function to the library: a "callback"

Callbacks:

Passing a function as a parameter

whenXHappens(function(x result){ do this with X })

Problem: callback hell: Doing several things *in turn* requires nesting!

"Solution": async/await. Fancy keywords that remove the nesting

For more: see "What Color is Your Function"

(Used in: Javascript, C#, Python)

Fibers

- Goals:
  - Divide the thread into several tasks
    - Independent tasks do their own work,
      as opposed to work being passed around as functions
  - Make it "look like" blocking
    - No "special" call syntax (callbacks, async/await)
- Multiple call stacks (one for each fiber)
  - Custom scheduler swaps between them
  - Custom assembly code for stack mgmt (just like in the OS)
- Cannot be mixed with blocking calls
  - A blocking call will block the fiber scheduler (and therefore all fibers)

Two ways to transform

Library solution

Rewrite the std. library to use non-blocking calls
and insert calls to yield

Can NOT mix and match libraries (including std. lib)

(Rust: tokio, D: vibe.d)

Language integration solution

Make the compiler transform blocking calls into nonblocking calls

Then insert calls to yield automatically after syscalls

Requires tight language integration:
Compiler needs to know how to transform OS interactions

(Go)

So far

- You can "layer" threading mechanisms
- You can not "mix" different mechanisms on the same "layer"
  - Pick *one* library, and don't mix it with the standard library
- Some languages do all the heavy lifting for you
  - Go is the best example of this

Interrupted when sharing

Side effects of preemptive scheduling

The demonstration

Thread 1: `i = i + 1;`
Thread 2: `i = i - 1;`

This is three operations:

- Read
- Modify
- Write

Swapping can occur at any time (with preemptive sched)

- The work of one can be overwritten by another
- Race condition: The result depends on ordering in time

Two solutions

Prevent things from happening at the same time

- Introduce a "bottleneck", where only one thread can act at a time
- This removes concurrency!
  This blocks other threads from working

Do not share resources

- Send messages instead of sharing memory
- This is a different "way of thinking"
  The problem must be "translated"

Bottlenecks

- Some sort of "indicator" that says if a resource is being used
  - If indicator says resource is in use, scheduler finds something else to do
  - Modifying this indicator must not introduce another race condition!
- Flag modifications must be indivisible
  - An atomic operation
  - Implemented using hardware instructions
    - disable/enable global interrupts (this disables the preemption in the scheduler)
    - compare-and-exchange instruction (CMPXHG)
    - memory bus locking instructions (LOCK prefix, MFENCE)
  - x86 is very hard, don't worry about this
    - The point is: it's not magic

Counters

Semaphore (aka counting semaphore)

Integer flag with value >= 0

Two operations:

signal: increment by one

wait: decrement by one

Can not decrement if the value is zero - blocks

Thread(s) will be awoken if someone else signals

Alternate names:

wait: P / Prolaag / Probeer te verlaagen (try to lower)

signal: V / Verhogen
notify (to indicate that control is not transferred to the waiter)

Flags

Binary semaphore

- Can only have values 1 or 0
  - Available / unavailable
  - Locked / unlocked
- Anyone can have the key!

Mutual exclusion (Mutex)

- Binary semaphore
- Requires that *only* the one that locks can unlock
  - No "unlocking on behalf of"
  - Think (temporary) "ownership" of the resource
- With ownership we can also provide priority inheritance
  - Taking the resource away from someone important gives us a priority boost so we can give it back sooner

Granularity

How many locks?

For how long do we hold them?

"Course-grained" locking:

- Extreme: One central database with one giant lock
- Can dramatically reduce the amount of concurrent execution

"Fine-grained" locking:

- Extreme: One lock per resource
- Can quickly get out of hand,
  especially when we need multiple resources at the same time

Sample C code

Mutex

```c
#include <pthread.h>

int main(){
    pthread_mutex_t mtx;

    // 2nd arg is a pthread_mutexattr_t
    pthread_mutex_init(&mtx, NULL);

    pthread_mutex_lock(&mtx);
    // Critical section
    pthread_mutex_unlock(&mtx);

    pthread_mutex_destroy(&mtx);
}
```

http://pubs.opengroup.org/onlinepubs/7990989775/xsh/pthread.h.html

Semaphore

```c
#include <semaphore.h>

int main(){
    sem_t sem;

    // 2nd arg: 0 for threads, 1 for proc's
    // 3rd arg: initial value
    sem_init(&sem, 0, resource_size);

    sem_wait(&sem);
    sem_post(&sem);

    sem_destroy(&sem);
}
// also: sem_getvalue, sem_trywait
```

https://pubs.opengroup.org/onlinepubs/7990989775/xsh/semaphore.h.html

Sharing uninterrupted

Sending messages to share values
instead of sharing memory to send values

Message passing

- Instead of sharing memory,
  we can send messages
  - Messages contain values
  - Often these values are copies of the senders' local data
- The receiver of the message performs actions
  - The action depends on the message contents (and local state)
  - Each thread is responsible for its own data
    - that's why we often refer to them as "processes"
      conceptually they don't share memory, though *technically* they do
- Share memory by communicating, instead of communicating by sharing memory

Taxonomy

Synchronous vs asynchronous:

- Sender must wait until receiver can receive
- "Phone" vs "Mail"
- Buffers are asynchronous

Symmetric vs asymmetric:

- Receiver specifies where (which sender/channel) it is receiving from
- "Instant messaging" vs "email"
- Channels are symmetric, Mailboxes are asymmetric

The queue

A thread-safe queue
is the most basic system

- Threads can insert elements and take them out safely
  - (No accidental overwrites when inserting, or duplications when extracting)
- Allows for multiple senders and receivers to use the same queue

A queue with one dedicated receiver is a mailbox

- Because there is only one receiver (the receiving thread owns its queue), you can usually not access the queue separately/directly

A mechanism to use multiple queues simultaneously makes them channels

- This mechanism is usually called select

Go channels

```
func make(t Type, size ...IntegerType) Type
// https://golang.org/pkg/builtin/#make

someChannel := make(chan T)          // makes a synchronous channel
someChannel := make(chan T, 5)       // makes a buffered (async) channel

make(chan T, 1)   - An interesting special case: See the mutex similarity?
```

```
func foo(someChannel <-chan int) {} // this function can only read from
                                    //   the channel

func bar(someChannel chan<- int) {} // this function can only write to
                                    //   the channel
```

Go channels

```
// Writing/sending
// Will only happen if
//  1) someone is waiting to read (unbuffered)
//  or 2) the buffer is not full (buffered)
someChannel <- val

// Reading/receiving
// Will only happen when there is a value to read
val1 int
val1 <- someChannel
val2 := <- someChannel
// Receive & discard. Useful when waiting for events
<- someChannel
```

Go channels

The correct way:

```
for {
    select {
        // Receiving
        case msg1 := <- chan1:
            // action 1
        case msg2 := <- chan2:
            // action 2

        // Sending
        // (Rarely needed. Use with caution - remember to send duplicates!)
        case chan3 <- msg3:
            // action 3, just after we sent something on chan3
    }
}
```

For-select loops

- Each process is responsible for its local data
- Messages are the boundary to the outside world
- Incoming messages represent events
  - Things that occur "once"
  - As opposed to polling - continuously checking
- Outgoing messages are (sometimes) sent in response
- Sources, sinks, servers
- main: create channels, create processes, sleep/log

You can send anything

- You can send pointers over channels
  - Slices (dynamic arrays) and maps (associative arrays) are pointers
  - Remember to duplicate / deep copy these *before* sending them
  - If you want channels, but automatic checks for this bug
    - Use Rust instead
- You can send channels over channels
  - Mostly useful for "send a reply back to me"

The message duplicator

```
func Repeater(ch_in interface{}, chs_out ...interface{}) {
    for {
        v, _ := reflect.ValueOf(ch_in).Recv()
        for _, c := range chs_out {
            reflect.ValueOf(c).Send(v)
        }
    }
}
```