8/22

Test Cases: Validity, Originality, Comment
- Print error if fails

OSDev.org

Different languages are good at different things — In so hungry :(
- We can't build an OS in Python

---

c = getch();
└ Reads next char and returns
- What does reading next char mean?
- Reads from standard input
  └ Choose file, keyboard, network connection to remote system, etc.
- You don't care about the source - abstraction so you don't care abt file vs input even tho completely different

Magic Piece
- Abstraction
  - Hide details - simplicity
  - Make things look the same - uniformity
- Resource Management
  - OS controls which programs get what memory, which program gets input, etc.
- Isolation / Protection
  - Program can't read another's files w/ correct permission
- Inter-process Communication
  - Processes share info w/ each other

Process - Running a program - putting it in memory
Program - Bytes on disk

UX - User Experience

- What non-programmers see - outside aspect - windowing, drag + drop, etc.

Connection btwn pieces and magic piece

---

c = getch();

- If user leaves what entering a char, what happens?
  - Has loop to check for char - wasting memory
    - Branch predictor becomes v efficient at running loop - fills pipeline w/ useless instructions, slows down/drains battery of system

```
while (no_char) {
...
}
```

- What to suspend a program
  - Pieces can have multiple things happening - concurrency
    - If waiting for input, other things can still be happening
    - Parallelism is a subset of concurrency
      - Concurrent things can run sequentially or parallel
  - Program can have concurrent things happen, system can have concurrent processes

- Concurrency - async/await
  - Async packages code into future and returns can evaluate/do w/

3 abilities

- Suspend one thing
- Maintain list of everything else to keep running
- Resume suspended thing - detect conditions to resume

---

getch(callback);
- Callback function
  - When you get a character, run callback function
- Very simple concurrency
- Classic JS requires callbacks
- Hard to see sequential program - hard to follow, lots of jumping around
  - Modern JS/Python give illusion of sequential - async/await compiles into callbacks
- If need to pass variable, would need to pass into callback parameter, callback needs to pass into other callbacks w/in itself

To share a variable, compiler needs to capture var and pass it around
- Need context of function - environment

Closure - Piece of code + environment - variable it needs
- Python - lambda, nested functions (functions inside functions)

---

int x = ...;
getch([x](char ch){ }) - Using closures
- [] w/ nothing to left means its a closure
- [x] means telling compiler to capture value of x
  - Shallow copy only of value of x
  - Can capture x by using &x - use pointer instead
    - Dangerous b/c memory at address x might get mapped to something else (x might not live long enough)
- Still looks ugly, its + lamp sucks, lots of nesting closures

Rule: Don't pass pointers or references unless you're convinced pointer will live long enough

Future<char> c = getchar(); - want non-blocking can survive what

- Instead of returning char, returns another data structure called a future
- Queues up functions so runs in correct order (if have 3 getch() functions)

Taking piece of work, package up into data structure, place into queue

└ All ways of implementing getch() are doing this in essence

8/24

Don't have standard C++ library since running directly on hardware
- Std lib come from kernel from OS, we're creating OS

C++ is a combination of Java and C
- Method vs functions
  - Methods bound to data, come from classes
  - Functions are pure code

```
struct ABC {
    int a;
    float b;
    void m() { }
};
```

struct vs class
- everything private by default in class
- everything public by default in struct

```
void func() {
    ABC x;
}
```
- In C++, x is not a pointer - it's an actual instance of ABC

methods don't take space in a struct

```
void func() {
    ABC *x = new ABC();
}
```
- Pointer to ABC, creating space on heap for it

Can use auto instead of type

```
class ABC {
    ABC() {...} - Constructor
    ABC(double z) {...}
}
```

ABC x{}; - initializing using default constructor → Not creating pointer
ABC p{20.0};

would be destroyed after function end

auto p = new ABC(20.0) - Creating ptr, created in heap

can use delete p;

new == malloc
delete == free

```
class ABC {
    ~ABC() {...} - Destructor - Called when delete used/object is destroyed
}
```

```
class Thing {
    int v;
public:
    Thing(int v) {
        this.v = v;
    }
}
```

=>

```
class Thing {
    int v;
public:
    Thing(int x) : v(x) { }
}
```

Becomes a rule: initializer of v equals x

same as left side, just more efficient code

```
class Thing {
  ...
  int operator +(int z) { return v + z; }   <- Overload + sign
};

Thing a{100};
print(a + 20); - prints int
```

```
class Thing {                overload function call, obj pretends to be function - returns int
  int operator()(char c) {...}
};

Thing a{100}
a('a');
```

```
int v = 100;                 Closure which captures v - will not run, stored to stack
auto a = [v](char c) { print(c + v); };      <- need to copy to heap

...

Later calls  a('x')
```

Compiler creates class which stores captured vars, overloads () operator to store body of lambda/function call

Closures passed by value

getch([v](char c){...}) - Closure inside copied into getch, original destroyed since only stored in stack

void getch( )
takes closure, but we don't know the type of a closure

Use generics in Cpp w template
↳ function generator

```
template<typename T>
T silly(T a, T b){ return a+b; }
```

```
template<typename T>
void getch(T closure){... closure('x'); }
```

thread - chunk of sequential code that runs concurrently w other threads
- share global variables and heap
- have own private variables (unshared)
- template, takes lambda as argument

```
thread([]{...});
```

Mental model:

thread() function returns immediately, continues to run separate code

- code inside lambda also runs, runs concurrently
  - can be in parallel, whenever - not sure

How to implement thread?

Queue of things to run

8/29

KernelInit called by one core, wakes up the rest

Hypervisor flag - Detects if running on real hardware or VM

thread([]{}) - Runs concurrently w/ whatever calls it
- Both running at the same time

Kernel - Piece of software - first thing to run on system
- Most fundamental piece of OS
- Only 1 per system - Manages threads

Core - Physical piece of hardware to execute instructions

Thread - Software abstraction, virtual - core) create threads to run concurrent code
One core can call thread and another can run the work inside thread

---

```
void KernelMain(void){
1.      printf("Start")
2.      thread([]{
I.          printf("Mid")
II.         printf("Mid 2")});
3.      printf("End")
```

Within one thread, 1 → 2 → 3
- Inside the inner thread, I → II
- 1 will happen before I
- However, we don't know if 3 will run before or after I / II
- 2 will be called before I (thread must be created first)

yield() - Thread says another thread can take one core
- Scheduling point - Can come back to run on another core

Hyperthreading - Core can run multiple hardware threads
- In our world, one core can only run one thread at a time

stop() - give someone else a chance to run
- Same as yield but doesn't come back

```
void stop() {
    while (true) yield();
}
```

- Very bad implementation, keeps running, doesn't free memory
- Functions correctly, just wasteful of resources

```
void thread(Work work) {
    work();
}
```

- Makes same core run work(), removes parallelism
- Every core needs to do work

↳ Create queue to store threads to run
- Ready Queue - Threads ready to run but not running
- What's the best way to schedule threads? Priority?

Queue has thing to go into queue type of lock for queue

Better to define global variables in .h file using extern, defined in .cc file
- Preprocessor substitutes includes textually - if defined in .h, multiple definitions
- Static limits access to only one file

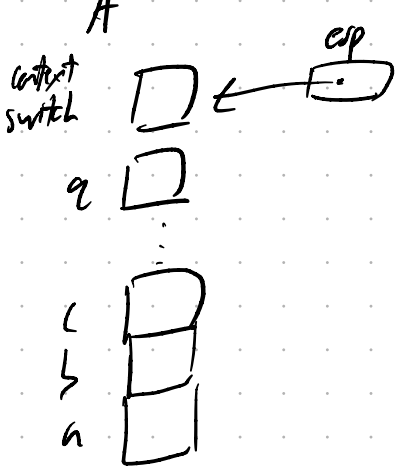
Control Block - Block of memory to control things
- Called TCB for threads

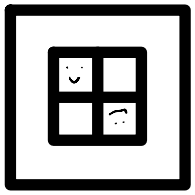
Think hard about using address - data can get destroyed
↳ Use TCB* tcb = new TCB();

Link elements in queue together using next pointer in TCB

8/31

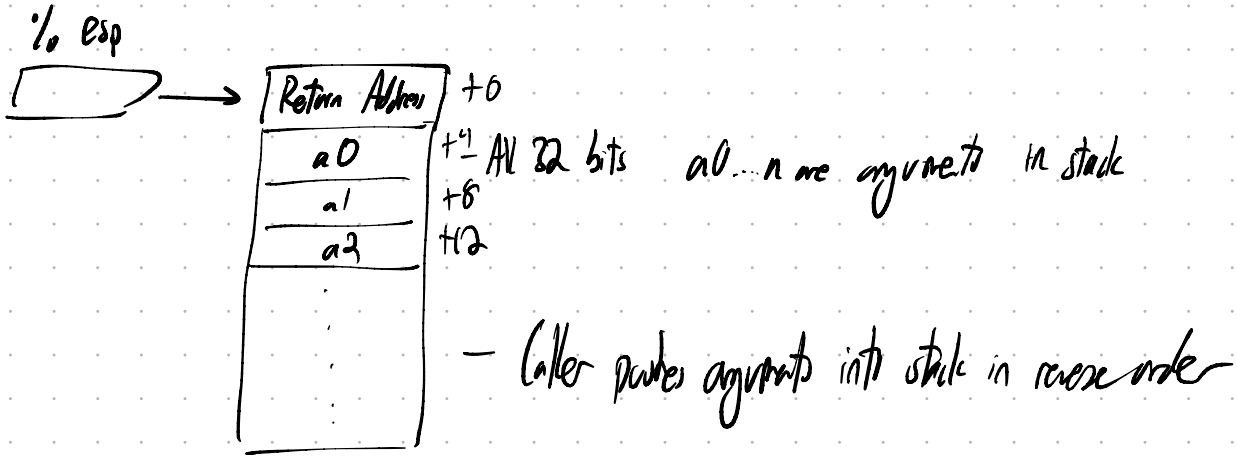
Needed to preserve context in thread switching
- Maintain pointer to top of stack





Threads share heap and global variables
- Only local variables are private
- Some registers as well

1. eax, edx, ecx, ebx, esi, edi, ebp, esp
   - Caller saved - Function called can use them
   - Stack pointer
   - Must be preserved, are callee-saved
   - 8 registers, ignoring float/double/vector registers
2. eflags



Can't have stack pointer saved into stack trying to preserve
- Would get updated to point to itself
- Instead, have context_switch() function take in pointer as argument
  - This is where to store the stack pointer

Every thread has its own stack

context_switch(uint32_t* save_esp_here, uint32_t* restore_the_esp_here):

```
                    // src          dst
mov 4(%esp), %eax       - Save stack pointer address (1st arg) to eax reg
mov 8(%esp), %ecx       - Save 2nd arg
// Using A's stack
push %ebx
push %esi
push %edi               - Save stack pointer into address given (address is in eax
push %ebp                   register)
mov %esp, (%eax)
// Saved my context

mov (%ecx), %esp
pop %ebp        <- In thread B now, takes prev arg and puts into registers
pop %edi        // using B's Stack
pop %esi
pop %ebx
```

ret

Expect stack to have 5 items: popped registers and return address
Stacks grows toward lower memory addresses
Return address is pointer to code

$100 = 101.4_8$

8/7

Lack of atomicity - Two cores can increment a variable at the same time, both set to the same value

Functions that start w __ - intrinsic function
- Defined in compiler

__atomic_fetch_add()
- Fetch (load) value and add to it
  - Do atomically so no one can come in between
- Processors run things out of order - A → B → C can be run as A → C → B
  - If B is atomic, has chance C will run first and load in value before B incremented due to speculation
  - Will observe changes in value in wrong order
  - __ATOMIC_SEQ_CST - Forces sequential ordering
- Weak memory ordering - If one core runs X → Y, other cores may not see it in order, can see new X and old Y be loaded out of order

```
write() {
    ready = true;
    val = 10;
}
read() {
    while (!ready);
    print(val)
}
```

→ Will not work be race condition → print val before set

```
write() {
    val = 10;
    ready = true;
}
read() {
    while (!ready);
    print(val)
}
```

Still might not work be assumes strong ordering
- Compiler might speculate ready → true, reads old value of val before changes

Enforce a fence - ppl cannot see changes after fence before changes before fence happen
- Slows down system tens of cycles
- Barrier - close, used for coordinating thread, synchronization
- Fence for correct memory ordering

load_fence() - Make sure all loads before fence happen before loads after
- Only controls loads, bringing from memory

store_fence() - Make sure stores before fence happen before stores after
- Only writing to memory

> Usually need both

_ATOMIC_SEQ_CST - Do whatever you can to ensure sequential ordering

```
write() {

    some_value = 10;
    store_fence();
    value_is_ready = true;
    //value_is_ready = true;
}

reader() {
    while (!value_is_ready);
    load_fence();
    print(some_value);
}
```

add_fetch - add, then return new value
fetch_add - get old value then add

SpinLock

```
SpinLock {
    Atomic<bool> taken;
    SpinLock() : taken(false) {}
    SpinLock(const SpinLock &) = delete;
    void lock(void) {
        while (taken.exchange(true)) {
        }
    }
    void unlock(void) {
        taken.set(false);
    }
}
```

Need to know if I own the lock - If I set false → true or alr was true

↳ Exchange - set to value, return old value

Use atomic

```
x = 10;
l.unlock();   - Guarantees sequential
y = 20;
```

LockGuard exists to protect against forgetting to unlock, leave what unlocking - destructor to help delete

Issue: No fairness, many threads racing to get lock could cause starvation - one thread could never get the lock

Bigger issue: Lock() function clogging core it's running on wasting resources, other threads can run when I'm waiting

Issue: Single core and two threads

```
SpinLock lock {};
thread [] {
    lock.lock();
    yield();
    lock.unlock();
});
thread [] {
    lock.lock();
});
```

1st thread locks
1st thread yields to 2nd thread
2nd thread tries to take the lock
- Sits there spinning indefinitely bc lock never released

lock() also wasted bc branch speculation fills buffer, then needs to flush when actually set to false

Solutions

yield() inside while loop in lock()
- Lets others run, solves deadlock problem
- Problem: yield() uses a Spinlock (ready queue has a lock)
  - Becomes deadlock / runs out of memory

pause() instruction in x86
- Says slow down, helps save energy - Doesn't solve 2 threads problem

hlt() instruction in x86
- Stronger pause(), puts core in low-power state, stop fetching instructions until interrupt happens (timer, keyboard press, etc.)
- Way to save energy, not giving others a chance to run either
- Doesn't solve 2 threads problem

mwait() instruction
- Give memory address of something don't like
- Wait until its value changes, puts in low power state until value changes

To solve 2 threads problem - need on the lock that has yield()

Race Condition

```
void lock(void) {
    while (taken.exchange(true)) {
        mwait(&taken);
    }
}
```

If another thread unlocks here, mwait is waiting for the wrong value

Solution

monitor() instruction monitor(&taken)

- Say plan on waiting for memory location, tells hardware
- When use mwait(), registered interest w/ hardware - can detect before start waiting - if change before hits mwait(), change next mwait() to NOP()

```
void lock(void) {
    monitor(&taken);
    while (taken.exchange(true)) {
        mwait();
        monitor(&taken);
    }
}
```

Have another lock that uses yield()

```
void lock(void) {
    while (taken.exchange(true)) {
        yield();
    }
}
```

yield gives chance for anyone holding the lock to release

Problems:

- Unfair, could have starvation
- Inefficient - Still asking question repeatedly, just wasting time in between
  - Still wasting time

Solution

Add "blocked" status to ready and running status

Blocking Lock - Stop running completely until lock is unlocked

- New queue for blocked threads, don't run until unlocked
- Attach queue to lock
  - Have flag and queue as variables in lock
  - Allows us to stop running + checking for unlock
  - Solves maintaining the queue w/ our policy for popping/running threads

Create yield w/ queue to add to - called block(addr)

```
void unlock(void) {
    auto who_needs_to_run = waiting.removed();
    if (who_needs_to_run != nullptr)
        taken.set(false);
    else
        ready_queue.add(who_needs_to_run);
}
```

Runs at race conditions - what to do?
- Add SpinLock inside BlockingLock

```
void lock(void) {
    inner_spin_lock.lock();
    while (taken.exchange(true)) {
        inner_spin_lock.unlock();
        block()
        inner_spin_lock.lock();
    }
}
```

- Still has race condition as before
  - If add after block() - no one to unlock since context switch

↳ Pass SpinLock as argument to block(), after switch + safely inside block_queue, then unlock

```
void unlock(void) {
    inner_spin_lock.lock();
    auto who_needs_to_run = waiting.remove();
    if (who_needs_to_run != nullptr)
        taken.set(false);
    else
        ready_queue.add(who_needs_to_run);
    inner_spin_lock.unlock();
}
```

waiting-q 0 taken true

1