Operating System(OS) support

October 14, 2015

Overview

- Hardware/OS architecture (296-300)
  - protection: user and kernel levels (301,
  - virtual memory and memory pages (302-304,307)
- Processes and threads
  - Process/thread creation (305)
  - OS threads vs. green (user-level) threads
    - blocking IO
- Thread programming
  - locks and monitors
  - mutual exlcusion and visibility problems
    - Java Memory Model (JMM)
- 7.6 and 7.7 are not covered (you may read them if interested)

OS architecture

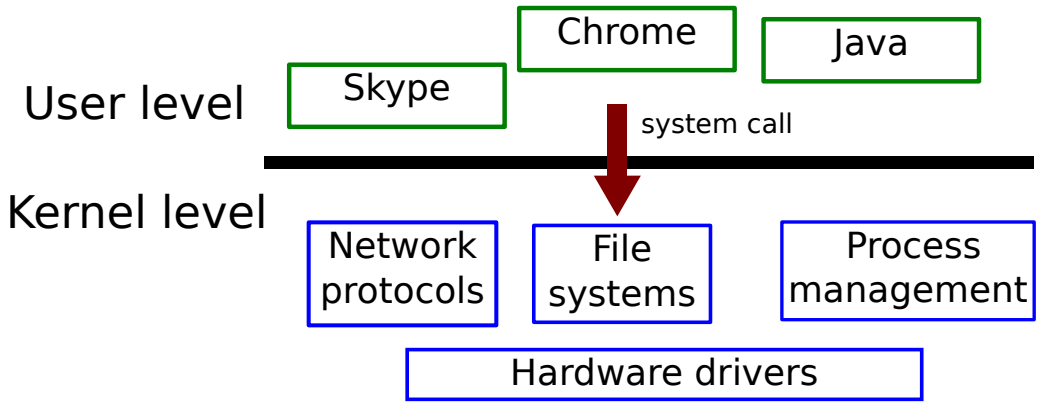
Why?

Why do we need to know internal workings of OS?

It can help to understand:

- how OS processes and OS threads work
  - how we can have 4GB of resident memory (RAM) and programs allocating 40GB of memory
    - what is virtual memory
  - how to share memory between processes
  - what is involved in sending a network packet
- why and where green threads are better than OS threads
  - understand what is context switch
  - why it is difficult to write pre-emptive green threads
    - why green threads require async IO

Overview



- interface to hardware (CPU, memory, disk, network card, monitor)
- core services (file systems, network protocols, ...)
- manage resources
  - memory management (virtual memory, memory pages)
  - CPU “management” (processes, threads)
  - IO resources (sockets, opened files)
- share resources between processes
  - *process* is a running program with its set of allocated resources
  - protect processes from each other

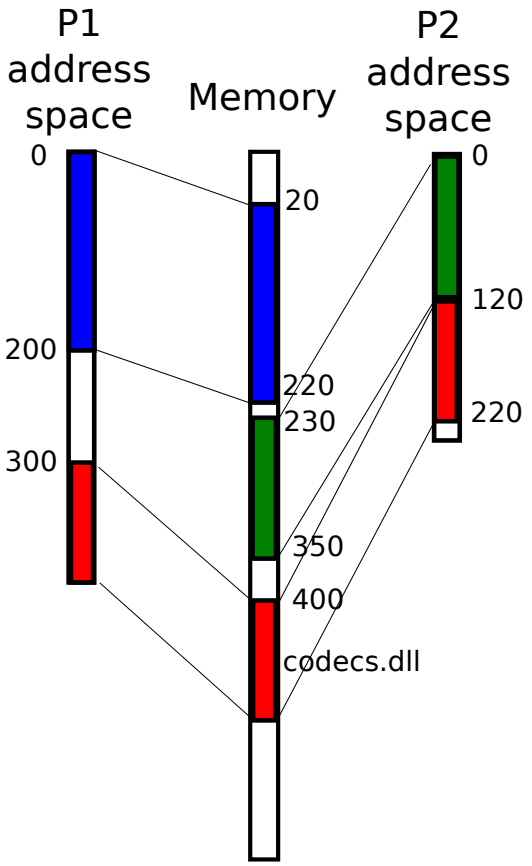
User and kernel levels

- A code may be running in different *Privilege levels*:
  - *kernel level* (tier 0) allows to access any resource,
  - *user level* (tier 3) restricts access to resources
    - programs mostly run in user mode, (try `time <program>`)
    - programs need to switch to kernel mode to access resources.
- *System calls* are kernel "functions" to be called from user level
  - e.g. `open()` system call

    ...returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (read(2), write(2)...
  - http://man7.org/linux/man-pages/man2/syscalls.2.html
- Memory - the resource a program can access without syscall
  - however, it does not access physical memory but *virtual memory*
  - mapping from virtual to physical is still controlled from the kernel

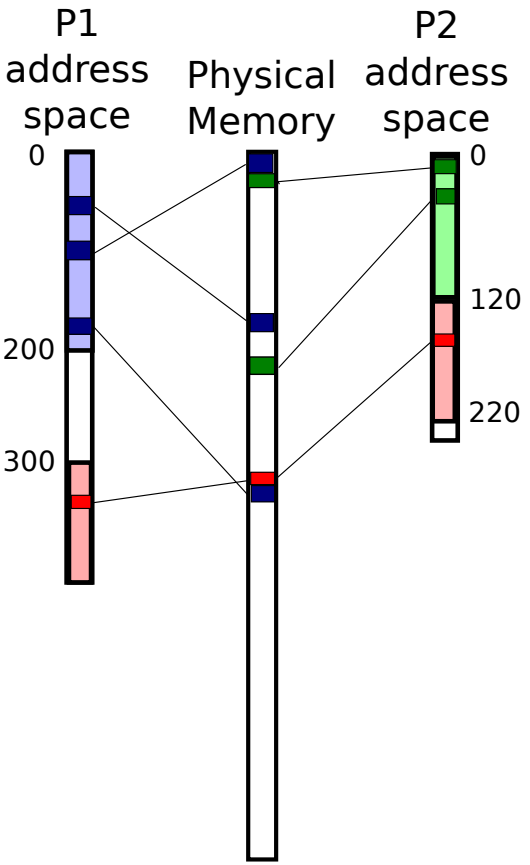
Virtual memory

- Every process has its *address space*
  - e.g. address 42 on processes A and B is different
- Address space is split into regions:
  - text (code), stack, heap
  - memory mapped files
    - e.g. dynamic library (DLL) code
  - shared memory regions
  - try `cat /proc/<PID>/maps`
- Problem: fragmentation
  - if P2 exits, can we fit something into the green region place in memory?
  - solution: *paging*



Memory pages

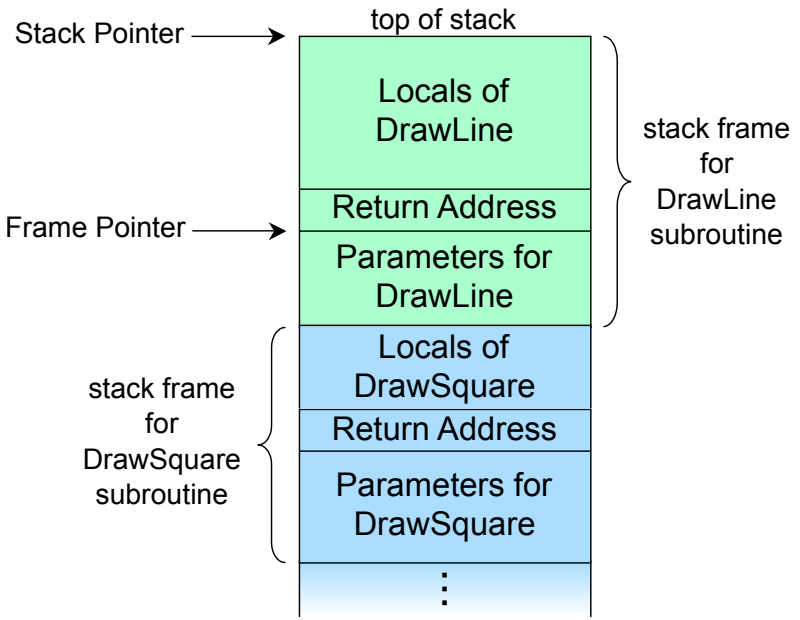
- mapping virtual (VM) to physical memory (PM)
  - pages are 4kB (4MB) chunks
  - process *Page Table* contains the mapping
  - TLB cache speeds up translation
- allocating VM region does nothing to PM
  - read/write results in *page fault*
  - page fault handler decides what to do
    - just allocate page in PM and update process page table
    - load data for memory mapped files
- in case of low memory some pages are offloaded
  - read-only mapped files can be discarded
  - data pages are written into swap
- VIRT,RES and SHR in top



Process context

Typical process consists of:

- currently executing command (Program Counter)
- Stack Pointer (SP)
- general purpose registers, floating point registers
- memory management registers
  - point to the information about process memory regions and page tables



Context switch

- Scheduler tries to run M processes on N cores
  - gives each process time slice to execute
- Switching - can only be done in kernel (privileged) mode
  - requires a system call
    - *pre-emptive*: as the result of CPU hardware exception
    - side effect of system call, e.g. in blocking in `recv()`
  - store old process context
  - find new process to execute (kernel contains list of processes)
  - load new process context (where it left the last time)
  - return to user mode
- TLB cache is invalidated (not with threads)
- CPU caches are also largely invalidated (unless threads reuse locality)

All in all, it is pretty expensive action

Creating processes

New processes are created with two system calls:

- `fork()` system call creates almost exact copy of the current process
  - information about memory regions is copied
  - page tables are copied
    - however, physical pages are not copied immediatelly
    - *copy-on-write*
- `execve()` system call runs another program in the current process
  - memory regions are cleared and replaced from the new executable
    - new code, data, stack regions, opened files and libraries
  - page tables are discarded
    - new information is filled as the program is loaded into physical memory

Creating threads

Thread - in Linux OS Light-Weight Processes (LWP)

- in Linux `clone()` system call creates new thread
- similar to process creation but shares with the parent process
  - paging tables and hence works on the same memory
    - this requires synchronization with other threads to protect the data from concurrent access
  - opened file descriptors

Look at fork(), execve(), mmap(), clone() system calls

- can you now roughly understand the manual for the functions?

Green threads

A.k.a. *user-space threads* or *library threads*

- Run completely in user space using one or several OS threads
  - *coroutines* – set of functions that are executed one after another
  - *fibers* – execution stack is captured during some calls (yield,send,alloc)
    - save stack: `setjmp`/`getjmp`, `setcontext`/`getcontext`
- Pros: much faster to switch than OS threads
  - do not need context switch or even system call
- Cons:
  - cannot utilize hardware cores (unless using several OS threads)
  - hard to do true pre-emption: user level does not have CPU hardware access, so have to ask kernel (i.e. system call)

Blocking IO with green threads

- what if: couroutine does not return, fiber never calls special function
- Lets have 10 couroutines running on 1 OS thread
  - one coroutine calls `socket.recv()` that blocks
    - OS thread is suspended inside kernel
    - interrupt it, it may break application logic
  - your other 9 coroutines have no chance to run, although 1. coroutine does not do useful work either
- Solutions:
  - create separate OS thread for coroutines that may block (Clojure)
  - use asynchronous IO, provided by the OS, to emulate synchronous IO for coroutines/fibers (Erlang)

Take your favourite language and find green (userspace,library) threads implementation. Find out how it switches (coroutines/fibers) and what it does with blocking functions.

Answering questions

Can we answer some of the questions on the Why slide?

- how OS processes and OS threads work
  - how we can have 4GB of resident memory (RAM) and programs allocating 40GB of memory
    - what is virtual memory
  - how to share memory between processes
  - what is involved in sending a network packet
- why and where green threads are better than OS threads
  - understand what is context switch
  - why it is difficult to write pre-emptive green threads
    - why green threads require async IO

Programming threads

Writer exclusion

- need to protect *shared state*

```
int i = 5;
public void increment() {
    i = i+1;
}
```

- 2 threads may call this code concurrently:
  - thread A reads value 5 and is pre-empted by OS
  - thread B reads value 5, increments and writes back 6
  - thread A waked up, it increments and writes back 6
  - two calls result in single increment (5->6)
- the problem: other thread starts reading and modifying the same data we are working on
- the solution: do not allow others to *read+modify+write* the data until we have changed it
  - if somebody just wants to *read* the data, it may be ok...

Reader exclusion

- need to protect *compound state*

```
int size = 0;
int array[] = new int[10];
public void insert(int x) {
    size = size + 1;
    array[size-1] = x;
}
```

- 2 threads may call this code concurrently:
  - thread A increments size and pre-empted
  - thread B tries to read the last element `array[size-1]` but there is no real value yet
  - thread A wakes up and writes `array[size-1]` but it is too late
- problem: operations on compound state must be *atomic*:
  - other threads should not see intermediate state
- solution: do not allow others to *read* the compound state until we have changed everything

Visibility problem 1

- It may seem we fix the last problem by rearranging two rows:

```
int size = 0;
int array[] = new int[10];
public void insert(int x) {
    array[size] = x;
    size = size + 1;
}
```

- it is **false**!!!
  - compiler and/or CPU may re-arrange the operations if different order is more efficient
- *visibility problem*: there may be no strict guarantee in which order updates are seen in other threads
  - Java Memory Model (JMM) makes such guarantees
- solution:
  - compiler does not re-arrange operations on some cases
  - CPU has special *fence instruction* (barrier instruction) not to re-arrange instructions over it

Visibility problem 2

- Imagine one thread doing something useful:

```
boolean running = true;
public void run() {
    while (running)
        doSomething();
}
```

- other thread tries to stop it:

```
running = false;
```

- compiler/JIT scans `doSomething()` and realizes it can keep `running` variable in register
  - the first thread never sees the update!
- problem: compiler may decide to keep variables in registers
- solution: instruct the compiler not to do that and read/write from/to memory (volatile in Java and C)
  - volatile in Java also allows not to reorder operations

Race condition

- Notice that some examples are quite unlikely to happen
  - this depends on the (unlucky) scheduling of threads
  - writer exclusion, reader exclusion, visibility problem 1
  - nevertheless, they may happen
- *Race condition* – unlikely and unlucky operation ordering due to timing (scheduling), that results in an inconsistent program state (bug)
- Imagine:
  - your program may work for 1 month with no problems
  - one day customer submits a bug
  - you cannot reproduce it, whatever you try
  - you may spend weeks but likely you answer will be “stupid user”
  - everyone will get angry
  - bug will still be there, to return to you in a week :)
- Race conditions tend to happen in a loaded (production) environment
- Bugs because of race conditions are the most unpleasant

Locks

- *lock* solves the exclusion problem (hence a.k.a. *mutex*)
  - lock can be taken and released
  - only one thread is permitted to hold it, other threads must wait until it is released
- Java synchronized(T) block:
  - on enter takes *intrinsic lock* of object T
    - in Java every object has single internal lock
  - on exit releases intrinsic lock of object T
- Allow only one thread at a time in writer exclusion problem:

```
int i = 5;
public void increment() {
    synchronized(this) {
        i = i+1;
    }
}
```

- same as `public synchronized void increment()...`

Efficiency of locks

- Lock is contended if many threads try to access it at once and fail
  - they have to go to sleep and hence switch the context
- lock first works through CPU cache, using Compare-and-Swap (CAS) operations
  - if this does not succeed, it asks OS to help it
    - makes system calls
- Hence uncontended locks are nowadays pretty efficient
- Contented locks are not

Java memory model (1)

- since Java 5 (1.5) there is Java Memory Model (JMM) specifications
- it makes very clear *visibility* and *ordering* guarantees using *happens-before* relationship:
  - **Program order rule**: each action in a thread happens-before every action in that thread that comes later in the program order
  - **Volatile variable rule**: A write to a volatile field happens-before every subsequent read from that same field.
  - ... 6 more rules
- Example. If thread A writes to y before thread B reads it:

```
volatile int y=0;
int x=0;
```

Thread A	Thread B
`x=5`	`int z=y;`
`y=3`	`System.out.print(x);`

- then thread B must print 5, because ordering guarantees are x=5 -> y=3 -> z=y -> print(x)

Java memory model (2)

- Other rules:
  - **monitor lock rule**: release of a lock happens before every subsequent acquisition of the **same** lock
- compiler and JVM may still re-order!
  - but they make sure it appears as if it was in the order specified by happens-before relationship
- Example:

```
x = 5
fun(x)
```

  - if compiler/JVM **knows** that:
    - fun() code has no guarantees about happens-before to other threads (no synchronizations, no volatile reads or writes)
    - it may unwrap fun() code and reorder operations, so that the result is still as if with program ordering

Example: double-checked locking

- Lets use “smart” trick to define singleton

```
class Foo {
    private static Helper helper;

    public static Helper getInstance() {
        if (helper == null) { // try not to use lock
            synchronized(Foo.class) {
                if (helper == null) { // check again
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
```

- may seem ok
  - if helper variable is monotonic, i.e. only possible transition is null -> instance
  - a thread either sees NULL or the created object, if NULL it synchronizes

Example: double-checked locking (2)

- JMM makes visibility guarantees between two threads if both use synchronize/volatile, it may be possible that:
- thread A creates `helper = new Helper()`
  - JVM unwraps constructor code for optimzation
  - JVM re-arranges writes to object fields and to static variable `helper`

```
helper = Helper()
helper.field1 = 3
helper.field2 = 6
```

- thread B reads `helper`, it is not null, so it does not synchronize
  - but its fields are not yet initialized

https://en.wikipedia.org/wiki/Double-checked_locking

> *The pattern, when implemented in some language/hardware combinations, can be unsafe. At times, it can be considered an anti-pattern.*

Monitors

- if thread A reads value from a queue
  - but can only proceed if queue is not empty
- or thread B writes value to a queue
  - but can only proceed if the queue is not full
- it is lock with precondition – *monitor*

More on thread programming

- Latches
- Concurrent collections: concurrent queue
- Thread pools