Processes

Johan Montelius

KTH

2020

A process

What is a process?

... a computation

- a program i.e. a sequence of operations
- a set of data structures
- a set of registers
- means to interact with other processes or external entities

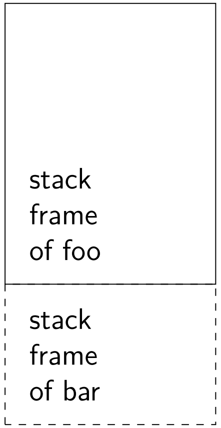
The C process

In C a process consist of:

- a program: a set of named procedures
- a call stack: provides the means to call and return from a procedure
- a heap from which we can allocate new data structures
- a current program pointer
- .... open file descriptors, etc

Understand how the call stack works and what the heap provides.

the stack



```
     int foo(int x, int y) {
───→   return x + y;
     }

     int bar() {
       int z;
       z = foo(3, 4)
       return z;
```

the frame

The *conceptual stack frame* contains:

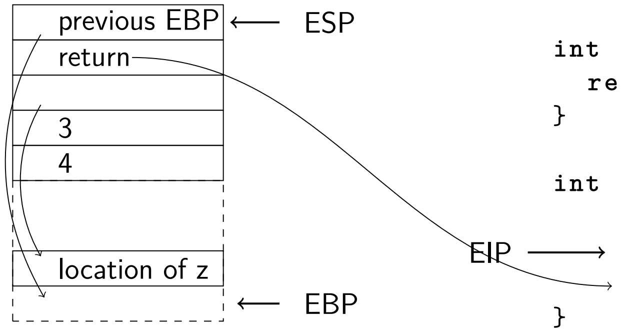
- the arguments of the procedure
- place to hold local variables
- magic information to be able to return from a call

Implementation of the stack

The CPU can be described by:

- the instruction pointer (EIP): a reference to the next instruction to execute
- a stack pointer (ESP): a reference to the top of the stack
- a base pointer (EBP): a reference to the current stack frame
- general purpose registers: used by a process to store data

the call stack



previous EBP ⟵ ESP

return

3

4

location of z

⟵ EBP

EIP ⟶

```
int foo(int x, int y) {
  return x + y;
}

int bar() {
    int z;
    z = foo(3, 4)
    return z;
}
```

… and more

The CPU has a set of general purpose registers. If a procedure uses these they should be restored when the procedure returns.

The general scheme shown is how the stack is implemented on a x86 architecture, other architectures have very different schemes.

Registers can be used to pass argument and to return values. The x86 architecture pass first six arguments and return value in registers.

Separate the *abstraction* of a C procedure call from how the stack is implemented.

a slight problem

How is a data structure returned from a procedure call?

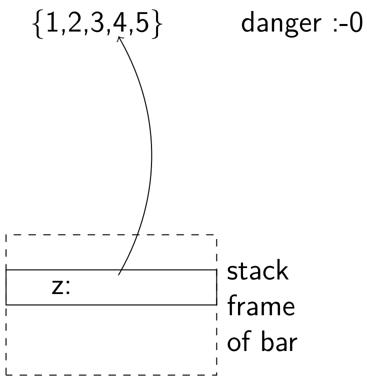
Three easy steps:

- the caller allocates room in its stack frame for the structure,
- the caller passes the location of the structure as an argument to the procedure and,
- the procedure updates the structure.

What if the caller does not know how large the structure will be?

Create a structure and return a pointer to the structure - problem solved.

the stack



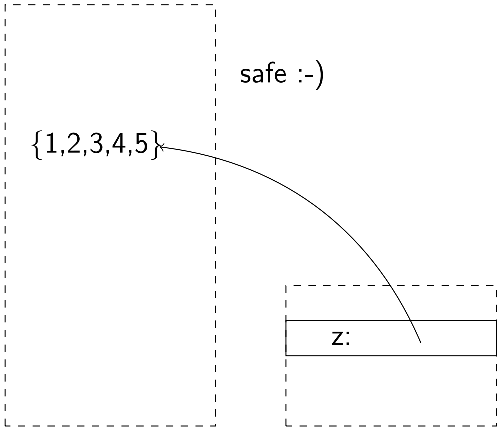
```
int *foo(int x) {
  int a[5] = {1,2,3,4,5};
  return a;
}


int bar() {
  int *z = foo(1);
  printf("z[2] is %d\n", z[2]);
  return 0;
}
```

not so good

We can not create structures on the stack and expect them to survive.

This is why we need the *heap*.

the heap



```
int *foo(int x) {
  int a[5] = {1,2,3,4,5};
  int *h;
  int i;

  h = (int*)malloc(5*sizeof(int

  for(i = 0; i != 5; i++) {
    h[i] = a[i];
  }
  return h;
}
```

the heap

- a memory area separated from the stack
- explicit allocation
- ... what about deallocation?
- the heap is handled using library calls in C

the heap API

- `void *malloc(size_t size)` : allocate `size` bytes on the heap, returns a pointer to the structure
- `free(void *ptr)` : deallocates a structure pointed to by `ptr`
- `void *calloc(size_t nmemb, size_t size)` : allocate an array of `nmemb` element of `size` bytes initialize it to zero
- `void *realloc(void *ptr, size_t size)` : changes the size of the structure pointed to by `ptr`

how about Java

```
public class RightTriangle {

    public double a, b, c;

    public RightTriangle(double x, double y) {
        a = x;
        b = y;
        c = Math.sqrt(Math.pow(x,2) + Math.pow(y,2));
    }

    public double area() {
        double ar = (a * b)/2;

        return ar;
    }
}
```

how about Java

```
public class Test {

    public static void main(String [] args) {

        RightTriangle egypt = new RightTriangle(3,4);

        double hyp = egypt.c;

        double ar = egypt.area();

        System.out.format("hypotenuse is: %.1f%n", hyp);
        System.out.format("   area is is: %.1f%n", ar);
    }
}
```

Java - objects on the heap

In Java *primitive types* and references to objects are allocated on the stack.

Regular objects are allocated on the heap explicitly using the `new` statement.

There is no explicit deallocation, memory is reclaimed by *garbage collection*.

.... A Java compiler can (sometimes) detect that an object will not live passed the point of a method return, and then allocate the object on the stack (escape analysis).

What about C++

In C++ it is up to the programmer:

- if nothing is stated an object is allocated on the stack (as in C),
- using the primitive `new` the object is allocated on the heap (as in Java).

There is no garbage collection in C++, objects need to explicitly deallocated using the primitive `delete`.

What about Erlang

All non-primitive data structures are allocated on the heap.

- Implicitly allocates data structures on the heap.
- The stack is only used for primitive data structures (integers, atoms, ..) and references to the heap.
- The heap, and thus the garbage collection, is per Erlang process.
- .... messages need to be copied from one heap to the other.

What about Rust

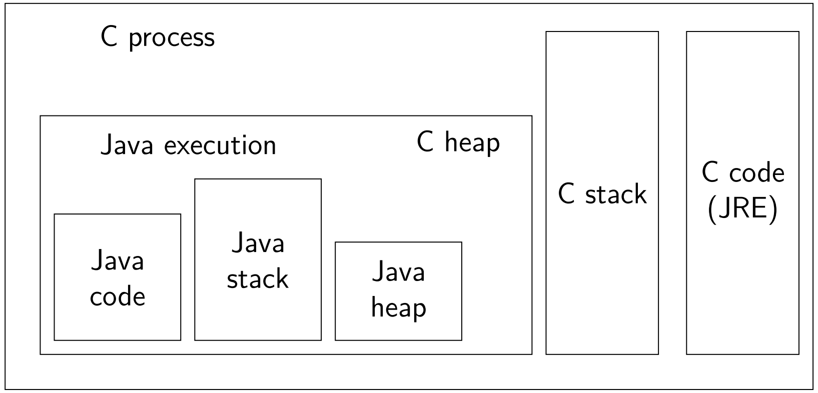
In Rust you get the best of both worlds.

- The programmer will explicitly create objects on the heap.
- The compiler knows when the object should be reclaimed.
- Rules of the language prevents you from doing mistakes.

The language prevents you from doing things that are possible in C.

let’s complicate things

The JRE, Java Runtime Environment, is an abstract machine i.e. a program that executes Java code. The JRE is typically implemented in C or in C++.



definition of C

Programming language constructs: variable declaration, procedure calls, for-, while-loops, etc.

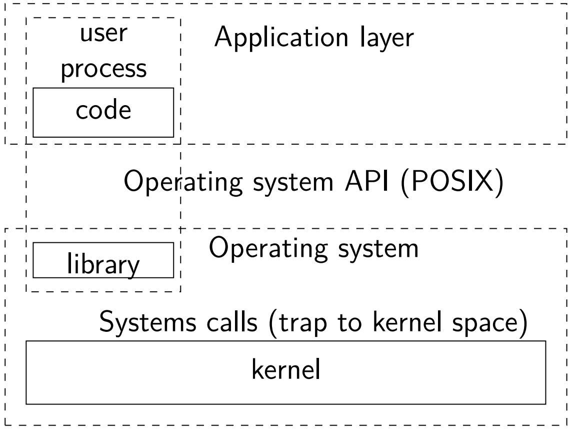
The C Standard Library (ISO C11)

- memory allocation: malloc, free, ...
- signal handling: signal, raise, kill, ..
- file operations: fopen, fclose, fread, fwrite, ...
- arithmetic, strings, ...
- ....

The POSIX operating system API

- process handling: fork, exec, wait, ...
- process communication: pipes, ..
- threads handling: pthread_create, ...
- managing directory and file ownership
- network handling: socket, listen, accept, ...
- ...

The operating system


user process
code
Application layer
Operating system API (POSIX)
library
Operating system
Systems calls (trap to kernel space)
kernel

Library is often just a wrapper for the system call - sometimes more complex.

the operating system

We will focus on how the operating system provides:

- means to create and start execution of a C process,
- a virtual execution environment that allows several processes to execute simultaneously,
- a persistent storage accessible to all processes,
- protection from unintended/malicious interference,
- yet allow them to share data and communicate.

note

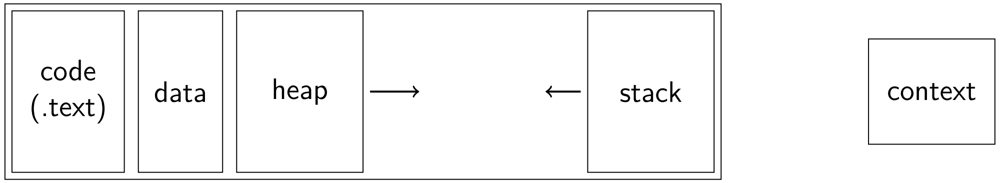
We're now presenting how the operating system **might** implement the *process abstraction*. One should never take for granted that things are actually implemented one way or another.

Examples are from Linux on a x86 architecture.

from program to process

How does the operating system create a process from a program?

- The program is found on some external data storage (hard-drive etc).
- Memory is allocated to hold the: code, static data, heap and stack.
- A *process context* is created holding the necessary register values etc.



process execution

How does the operating system control the execution of a process?

Indirect execution:

> The operating system decode and execute the instructions of the user process. This is done in Java, Erlang and many other languages based on a *virtual machine*.

Direct execution:

> The operating system loads the code of the user process, sets the stack and heap pointers and jumps to the first instruction of the process.

who is in control?

The operating system loads the code to memory, sets the register values for stack and heap pointers and ...

> .... sets the instruction pointer (EIP).

The future is now determined by the execution of the started process.

Is this a good thing?

limited direct execution

The *user program* is allowed to execute directly but is *limited* in that:

- it will not be able to execute all instruction
- it will only be able to access its own memory
- it will only be allowed to execute for a limited time

How do we implement these limitations?

The hardware allows an execution to be in either “user mode” or “kernel mode”.

life of an execution

Hardware - turn on power - start executing “BIOS”

Kernel mode: load operating system - set up *interrupt descriptor table* (IDT)

Kernel mode: load user program - set up stack, heap etc - start execution

User mode: run program

User mode: prepare system call - *trap to kernel mode*

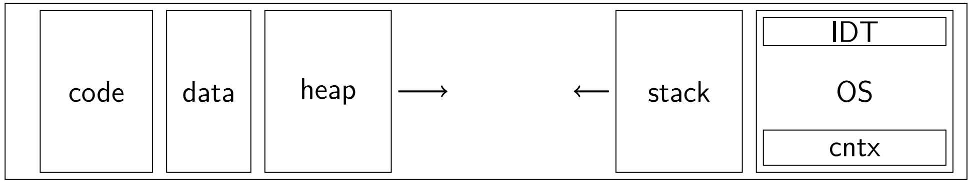
Kernel mode: *jump to system call handler*

Kernel mode: handle system call - prepare result - return from trap

User mode: continue execution

Important - the interrupt descriptor table must be protected, not modified in user

the process memory layout



code

data

heap ⟶

⟵ stack

IDT

OS

cntx

The operating system and user program is part of the same memory layout.

system calls in Linux on x86 - traditional

A user process will set the *system call identifier* in a specific register (%eax) and store the arguments on its stack.

An special instruction, `INT 0x80`, will set the execution in *kernel mode* and jump to the `0x80` position in the interrupt table (IDT).

The kernel will check the *system call identifier* and jump to the *service routine* specified by the *system call table*.

When the system call has been handled, execution returns to *user mode* using the `IRET` instruction.

The `Interrupt Descriptor Table` can only be set using the *privileged instruction* LIDT (*Load Interrupt Descriptor Table*).

system calls in Linux on x86 - today

The traditional system call implementation is expensive.

Some system calls, typically reading information, do not require the full protection of the switch from user to kernel mode.

Modern x86 processors provide more efficient support to handle system calls using the instructions `sysenter`/`syscall` and `sysexit`/`sysret`.

Check `vsyscall` *and* `vdso` *to learn more.*

user and kernel space

The kernel, the heart of the operating system, is executing in *kernel mode* (privileged mode, *ring 0*) where it has complete access to the hardware resources of the machine.

We implement *limited direct execution* by executing a user process in *user mode* (unprivileged, *ring 3*) were it will only be allowed to touch its own code, stack and heap (the *user space*).

The kernel should not take for granted that it can trust memory references from user space - security and portability. It should use special procedures when reading or writing to user space.

Take over control

If the kernel allows the user process to start executing, how does it stop it?

- The user process executes a INT instruction.
- The hardware generates an *interrupt*.

The kernel sets a *timer interrupt* before it allows the user process to execute.

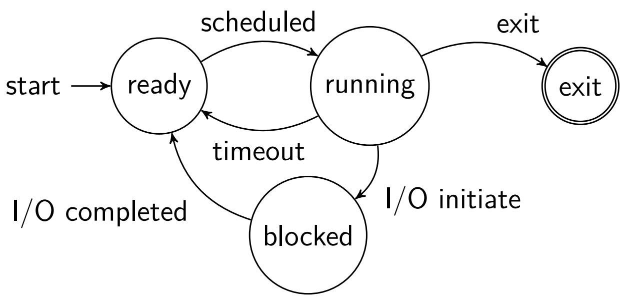
When the interrupt is generate by the hardware, the kernel can make a decision to *schedule* another process.

Interrupts

- Asynchronous interrupts: raised by the hardware: keyboard, IO, etc, can be *masked* by the kernel.
- Synchronous interrupts - exceptions: raised by the CPU
  - faults: something strange happens, will be handled by the kernel and instruction is re-executed (if the kernel can fix it).
  - traps: something special happens, used mainly by debugging.
  - abort: severe errors that will not be fixed by the kernel.
  - programmed exceptions (software exceptions): raised by for example the `INT` instruction, used by system calls and debuggers.

This is the Intel terminology.

process state



Where are interrupts used?

creating a process

How do we create a new process?

- A user process can create a new process.
- Functionality provided by a system call.
- In Unix the procedure is . . . strange, but very efficient.
- The POSIX API is not exactly what the Linux kernel provides - wrapper functions are used.

A knife, . . .

The `fork()` system call will create a new process - that is a copy of the current process.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

  printf("Let's go \n");

  int pid = fork();

  printf("  Hello, the pid is  %d\n", pid);

  sleep(10);
  return 0;
```

is the memory shared?

```
int main(int argc, char *argv[]) {
  int x = 42;

  int pid = fork();

  if(pid == 0) {
    sleep(10);
    printf("  Hello, I'm the child and x is  %d\n", x);
  } else {
    sleep(10);
    printf("  Hello, I'm the mother and x is  %d\n", x);
  }
  return 0;
}
```

magic

```
main(int argc, char *argv[]) {
  int x = 42;
  int pid = fork();

  if(pid == 0) {
    x = 12;
    sleep(10);
    printf("  Child:  address of x is %p\n", &x);
  } else {
    x = 13;
    sleep(10);
    printf("  Mother: address of x is  %p\n", &x);
  }
  return 0;
}
```

This will be explained when we look at memory virtualisation

what about open files

```
int main(int argc, char *argv[]) {
  FILE *foo = fopen("foo.txt", "w+");

  int pid = fork();
  if(pid == 0) {
    fprintf(foo, "  this is the child \n");
  } else {
    fprintf(foo, "  this is the mother \n");
  }
  return 0;
}
```

process creation

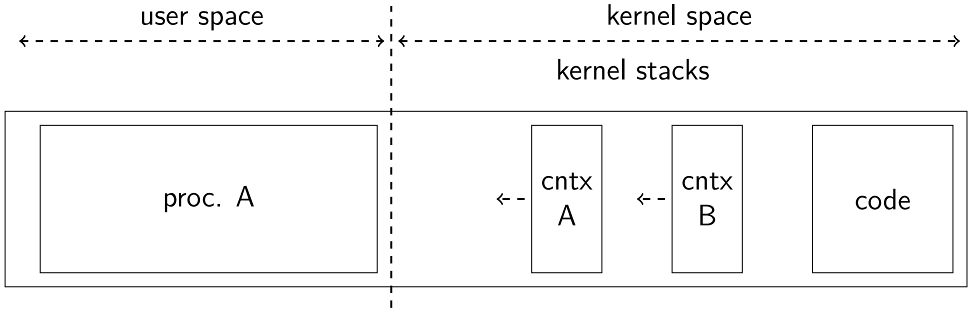
- The fork() system call will create a copy of calling process.
- The memory of the two processes are separated from each other (but use the same addresses).
- Already open files are shared by the two processes.
- Newly open files are not shared.

process API

The POSIX process API (part of):

- `fork()` : create a clone of current process
- `exit()` : terminate process
- `wait()` : wait for a process to terminate
- `exec()` : load and start execution of program
- `getpid()` : get process identifier
- `kill()` : send a signal to a process
- `raise()` : raise an exception
- `sigaction()` : sets a signal handler

process scheduling



user space

kernel space

kernel stacks

proc. A

cntx A

cntx B

code

what happens when a timer interrupt is received?.

switching context

When a timer interrupt is received:

- save the current registers in the context (*process table*) of the current process,
- decide which other process that should run,
- load the stored registers of the selected process,
- reset the timer, switch to user mode and,
- set the instruction pointer of the selected process.

The kernel also needs a stack and uses a per-process kernel stack.

scheduling

Next lecture:

- How to decide which processes that should be scheduled for execution.
- Read chapter 7, 8 and 9 in “Three Easy Pieces”.
- Do the assignments, they will help you understand the nature of processes.