

xv6©-rev10  
(Copyright Frans Kaashoek, Robert Morris, and Russ Cox.)  
The Process Model

Carmi Merimovich

Tel-Aviv Academic College

September 17, 2017

- The process model is determined by both: Hardware and Software.
- The process model is visible directly to Assembly language programmers.
- Higher (and higher) level languages masks the process model from the user.

## Address space

- Address width is hardware dependent. (32-bit on 32 bit Intel's).
- I.e., the address space is 4GB.
- The software can (and does) impose limits on user usage.
- (This required hardware assistance).

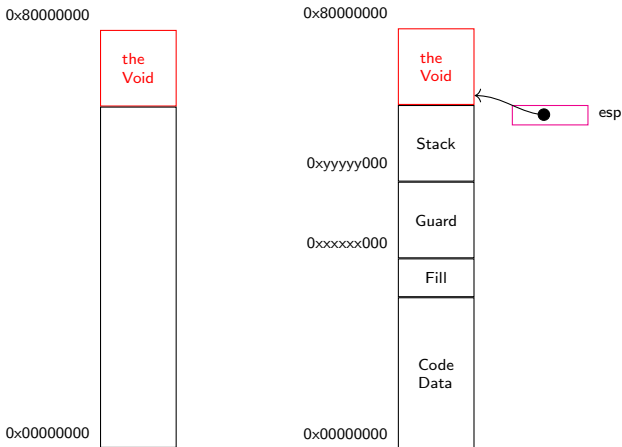
# Process Memory

- The memory can be continuous or discountinuous.
- Parts of it can be read-only.
- (Requires hardware assistance).
- The kernel enforces memory separation between processes.
- It does not care for process bugs.
- e.g., byte sized process vs. page size granularity.

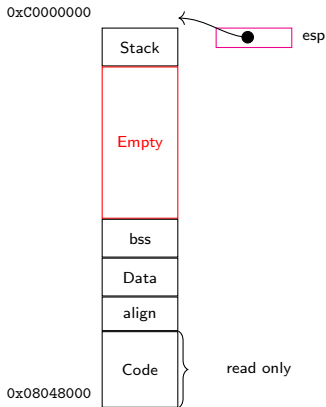
## xv6 process memory

- Program code/data begins at address 0.
- Memory is continuous.
- Memory can be in the range (0-7FFFFFFF) (i.e., 2GB).
- In fact only 1MB, but this is easily fixed.
- Stack is above the program.
- There is a guard page between the code/data and the stack
- Note a program can change the above.

## xv6 default process memory layout



# Linux default process memory layout



## xv6 process execution model

- Only non-privileged(!) instructions can be used.
- Integer registers: eax, ecx, edx, ebx, esp, ebp, esi, edi.
- Segment registers: cs, ds, ss, es, fs, gs.
- Of course: eip.

There are MANY MORE unprivileged registers.

- fp0-fp7.
- mmx, xmm, sse 1,2,3,4....
- They are NOT PART of the xv6 process model.
- However: There's no hardware mechanism enforcing this!
- It is the programmers problem if she uses them.



Can a process do anything other than calculations?!?!?!?!?

NO. Other stuff can be done ONLY by the kernel.

System calls or System services

## xv6 process related system calls

Table: Process control

|                                    |                                  |
|------------------------------------|----------------------------------|
| <code>fork()</code>                | Create process                   |
| <code>exit()</code>                | Terminate current process        |
| <code>wait()</code>                | Wait for a child process to exit |
| <code>kill(pid)</code>             | Terminate process pid            |
| <code>getpid()</code>              | Return current process id        |
| <code>sleep(n)</code>              | Sleep for n timer interrupts     |
| <code>exec(filename, *argv)</code> | Load a file and execute it       |

Table: Memory control

`sbrk(n)` Grow process memory by n bytes

## xv6 file related system calls

### Table: file name control

|  |  |
|--|--|
| <code>open(filename, flags)</code>     | Open a file; flags indicate read/write   |
| <code>pipe(p)</code>                   | Create a pipe and return fds in p        |
| <code>chdir(dirname)</code>            | Change the current directory             |
| <code>mkdir(dirname)</code>            | Create a new directory                   |
| <code>mknod(name, major, minor)</code> | Create a device file                     |
| <code>link(f1, f2)</code>              | Create another name (f2) for the file f1 |
| <code>unlink(filename)</code>          |  |

### Table: file control

|                                |   |
|--------------------------------|---|
| <code>read(fd, buf, n)</code>  | Read n bytes from an open file into buf |
| <code>write(fd, buf, n)</code> | Write n bytes to an open file           |
| <code>close(fd)</code>         | Release open file fd                    |
| <code>dup(fd)</code>           | Duplicate fd                            |
| <code>pipe(p)</code>           | Create a pipe and return fds in p       |
| <code>fstat(fd)</code>         | Return info about an open file          |

## system calls?!

- The above looks like a list of C callable routines.
- Which indeed they are!
- But then the code runs in the process model, hence can do nothing...
- First: Recall the C calling convention.
- Second: We see what is **really** a system call.

## gcc C calling convention: Caller

- Push to the stack arguments from right to left.

```
pushl   arg n-1
      ⋮
pushl   arg 0
```

- Invoke the callee with the **call** instruction.

```
call f-name
```

- Assumes the callee preserved registers except **eax**, **ecx**, **edx**.
- Removes arguments from stack.

```
addl $4*n,%esp
```

- Integer function return value is in **eax**.
- (Caller should save register **eax**, **ecx**, and **edx**, if it needs them).

## At&t syntax vs. Intel syntax

At&t:

```
addl $4*n,%esp
```

Intel:

```
add esp,4*n
```

1. Operand size is adjoined to the mnemonic.
2. Source and destination order is the opposite of Intel/MS order.
3. Immediate operands are preceeded by \$.
4. Register names are preceeded by %.

## Calling a 0-arguments functions

Listing 1: source

```
void main(void) {  
    extern fork();  
  
    fork();  
}
```

Listing 2: compiled

```
.globl fork  
main::  
    call fork  
    ret  
}
```

## Calling a 1-argument function

Listing 3: source

```
void main(void) {  
    extern close();  
  
    close(0);  
}
```

Listing 4: compiled

```
.globl close  
main::  
    pushl $0  
    call close  
    addl $4,%esp  
    ret  
}
```



## Calling a 2-arguments function

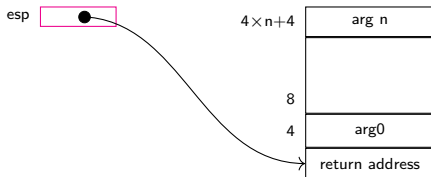
Listing 5: source

```
void main(void) {  
    extern open();  
  
    open("f",0);  
}
```

Listing 6: compiled

```
.globl open  
.code  
main::  
    pushl $0  
    pushl $s  
    call open  
    addl $8,%esp  
    ret  
  
.data  
s: .asciz "f"
```

## Stack on entering a routine

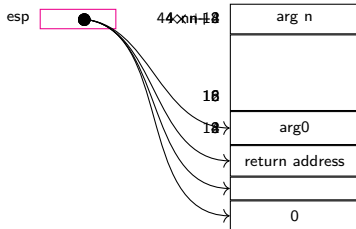


## gcc C callee

- Arguments are to be found on the stack above the return address.
- Registers **ebx**, **ebp**, **esi**, **edi**, and **esp**, should be preserved.
- (integer) Return value is put in **eax**.
- Returning using the **ret** instruction.
- Note: accessing the paramters without a frame pointer is error-prone for humans.

## Arguments offset relative to **esp**

```
movl 8(%esp),%eax  
  
subl $4,%esp  
movl 12(%esp),%eax  
  
pushl $0  
movl 16(%esp),%eax  
  
addl $8,%esp  
ret
```



## Arguments offset relative to **ebp**

```

pushl %ebp
movl %esp,%ebp

movl 12(%ebp),%eax

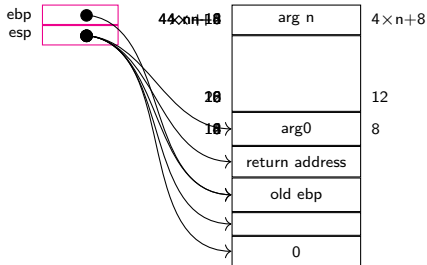
subl $4,%esp

movl 12(%ebp),%eax

pushl $0
movl 12(%ebp),%eax

addl $8,%esp
popl %ebp
ret

```



## system services

- Each kernel supplies a set of system services.
- These routines run at a privileged (i.e., kernel) state.
- Each routine is identified by a number (which is not very comfortable).
- Each routine might accept arguments.
- Each kernel/hardware combination has its own system call convention.

## xv6 fork/close/open system call

Listing 7: fork

```
movl $1,%eax
int $64
```

Listing 8: close

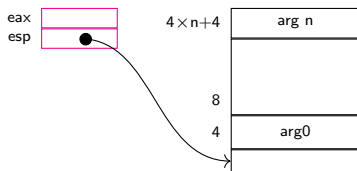
```
pushl $0
subl $4,%esp
movl $21,%eax
int $64
addl $8,%esp
```

Listing 9: open

```
pushl $0
pushl $s
subl $4,%esp
movl $13,%eax
int $64
addl $12,%esp

::
.data
s: .asciz "f"
```

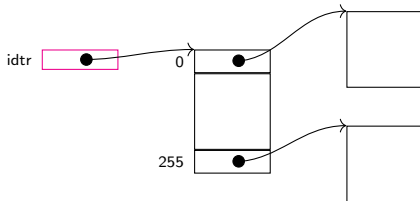
## State at **int \$64** execution





## int \$64 in the xv6 process model

- The **int** instruction is a glorified **call** instruction.
- It has one “**side effect**”: The processor switches to kernel.
- The operand (0–255) is the **number** of the function to run.
- Attempting number other than 64 yields a security violation.
- Outside of the process model:



## Demonstrating a different system call convention (Linux)

## Linux system (service) call convention

- The arguments are loaded into registers in the following order: ebx, ecx, edx, esi, edi, ebp.
- System call number is in eax.
- **int \$128** is executed.
- Success/failure code returned in eax.
- Except eax, registers (of the process model) are preserved.

## linux/x86-32 **fork/close/open** system calls

Listing 10: fork

```
movl $2,%eax  
int $128
```

Listing 11: close

```
movl $0,%ebx  
movl $6,%eax  
int $128
```

Listing 12: open

```
movl $0,%ecx  
movl $s,%ebx  
movl $5,%eax  
int $128  
:  
.data  
s: .asciz "f"
```

## Kernel defined xv6 process model

## xv6 Process model: files

- File is a general name which can refer to:
  - (disk) file.
  - pipe.
  - device.
- Mostly, files have names.
- The file names form hierarchy through folders.
- Names(!) can be added/removed to/from existing files.

## xv6 Process model: files

- The file supporting system calls are divided into:
  - Name only services.
    - Name beginning with '/' are considered absolute.
    - Names not beginning with '/' are relative to the **current working directory**.
  - Content services.
- In order to deal with the content of the file it should be "opened".
- Opening a file creates a context for it in the kernel.
- This context is referred to with the **file descriptor**.
- Up to 16 file descriptor can be active in parallel.