Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Operating Systems

Part 1: Virtualization – 1) Processes

Revision: `master@323240b (20230907-115823)`

BTI1341 / Fall 2023/24

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Outline

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CPU Virtualization

CPU Virtualization

Recall: In the introduction, we provided a few motivations to virtualize the CPU:

- Enables running multiple programs at the “same” time
- Provides the illusion of having an infinite number of CPUs
- Instructions from different programs do not interfere with each other

The interesting question now is: How can such virtualization actually be implemented?

Virtualization: Abstract Idea

The main concept behind virtualization is to provide access to a single resource multiple times “at once” (and probably for different parties).

In the physical world, this is difficult. However, in computing, we can resort to a trick:

- Each resource is available only once
- Full access is given for a resource…
- …but only during a *restricted* time frame
- This is known as time sharing
- Everyone requiring the resource accesses it in turns

How is this possible?

Introduction to Processes

The Process

Definition

A process [a] is (informally) a *running instance* of a *program*.

- A program is a set of instructions (and possibly data) stored *on disk*
- Each program in general exists only once on a computer
- A process is an instantiation of a program
- There can be $0 \dots N$ processes (from different programs) running at the same time

[a] Sometimes also called a task

Process as an Abstraction

A process is an *abstraction* of a running program: A representation of everything relevant being read or written while the program is running (its machine state). Most importantly:

- The address space : The whole memory belonging to the process
- CPU registers , especially
  - The program counter (PC) , also called instruction pointer (IP)
  - Stack pointer (SP) and corresponding frame pointer (FP)
- I/O information (e.g. open files)

Process API

In order to work with processes, an OS must provide an API, which supports in some form:

- The *creation* of processes, as well as their *destruction*
- Functionality for *waiting* and *controlling*
- Access process *status* information

All modern operating systems provide such an API, although they all look a bit different.

Process Creation

When creating a process, the OS performs a series of steps. In a nutshell:

- Load the program into memory:[1] executable *code* and *static data* (i.e. initialized variables)
  - This includes code from *shared libraries*
- Allocate the stack ; often initialized with *program arguments* and *environment*
- Maybe preallocate some heap memory
- Initialize I/O
  - E.g. UNIX: open **`stdin`**, **`stdout`** and **`stderr`** file descriptors
- Run the **`main()`** function

[1] This is often done *lazily*, i.e. only when selected parts are required.

Process Creation Illustrated

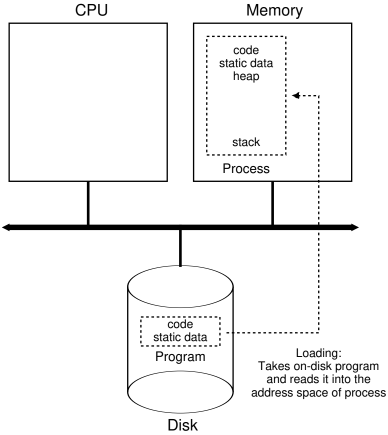

Figure: Loading a Program

Courtesy of [ADAD18]

Process States

A process can be in different states, simplified:

- **Running** The process is running on a CPU, i.e. *it is executing instructions*
- **Ready** The process is *ready to be run*. For some reasons, the OS has chosen not to run it at this given moment
- **Blocked (waiting)** The process has performed some operation which makes it wait for an event to happen. Often, this is caused by an I/O request: reading data from disk or waiting for user input.

We say that a process is being scheduled if it moves from ready to running state; descheduled if it moves from running to ready.

Process States Illustrated

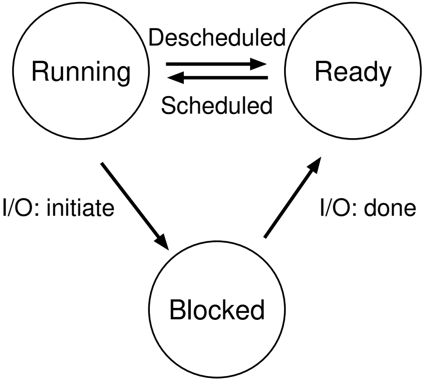

Figure: Process States and Transitions

Courtesy of [ADAD18]

Process State Trace: CPU only

This is an example of two processes running, using only the CPU (no I/O):

Time	**Process$_0$**	**Process$_1$**	**Notes**
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process$_0$ done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process$_1$ done

Process State Trace: CPU and IO

This is an example with two processes; at some time, one of them is blocked due to an I/O request.

Time	**Process$_0$**	**Process$_1$**	**Notes**
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process$_0$ initiates I/O
4	Blocked	Running	
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process$_1$ done
9	Running	-	
10	Running	-	Process$_0$ done

OS Data Structures

Internally, OSes use a variety of *data structures* to track and manage processes, I/O, users and much more. Typically, a process list (task list) tracks all processes present in a system.

A process structure (sometimes called process control block (PCB) or process descriptor) represents an individual process:[2]

- Process ID (PID)
- Parent process
- Process state
- Information about memory, stack, ...
- Context (register values)
- ...and more!

---

[2] The fields in the process structure are highly dependent of the OS. Different fields may be present and they may have different names!

The *xv6* `proc` Structure

Source: [xv6a]

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

Limited Direct Execution

Time Sharing for Virtualization

Now that we have some understanding of processes, let's come back to CPU virtualization:

- CPU virtualization can be achieved using time sharing
- *Basic idea:* let a process run for a while, then switch to another
- Issue No.1: *Control*
  - How to keep control over the CPU?
  - Why should a process return? What about endless loops?
  - How to prevent it from accessing restricted information?
- Issue No.2: *Performance*

Limited Direct Execution

The model which we will use is called limited direct execution . It consists of two key points:

Direct Execution: Processes run directly on the CPU, i.e. there is no additional abstraction layer between process and CPU (performance!)

Limits: There are some limiting mechanisms in place for

- Running time (we want to switch back to the OS or other processes...)
- Resource access (e.g. memory of other processes or I/O devices)

For this, some support from the CPU is required: Distinction of user mode and kernel mode .[3]

[3]★ Sometimes, these are also referred to as *CPU rings* or *domains*.

Direct Execution
(Without Limits)

OS	**Process**
Create process list entry	
Allocate memory	
Load program into memory	
Set up stack (`argc`/`argv`)	
Clear registers	
Execute **call** `main()`	
	Run `main()`
	Execute **return** from `main()`
Free memory of process	
Remove from proc. list	

Interlude: What is the Kernel?

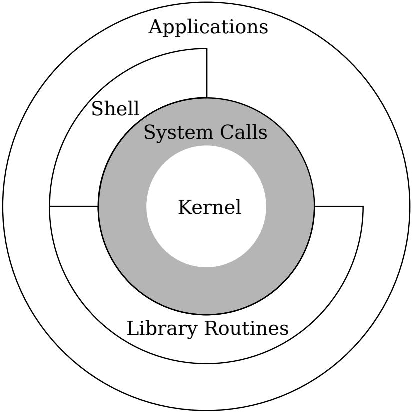

Figure: Architecture of the UNIX Operating System

Courtesy of [SR13]

Switching Between Modes

Having user- and kernel mode enables limiting the capabilities of a process. But two important questions remain open:

1. How can a process do privileged things, e.g. access a file?
2. How does the OS regain control?

Introducing System Calls

Question 1 is generally solved using system calls [4] (think of a "function call" from the process into the OS):

- With a special trap CPU instruction (often an interrupt), the process returns control to kernel mode. It's context is saved by the CPU on the kernel stack
- The CPU knows which kernel code to call due to a trap table (often called interrupt vector table), *initialized at boot time*
- Using a return-from-trap instruction, the OS returns control to the process, context is restored

---

[4] ☞ On GNU/Linux, you can obtain a list of system calls with "`man 2 syscalls`". In general, there is also a man page per syscall available with "`man 2 <syscall>`" – use it!

Limited Direct Execution

OS @boot (kernel mode)	Hardware
Initialize trap table	
	Store addr. of syscall handler

OS @run (kernel mode)	Hardware	Process (user mode)
Create process list entry Allocate memory Load program into memory Set up *user stack* (**`argc`**/**`argv`**) Push regs/PC to *kernel stack* **return-from-trap**		
	Restore regs from *kernel stack* Move to user mode Jump to `main()`	

Limited Direct Execution (cont.)

OS @run (kernel mode)	Hardware	Process (user mode)
		Run `main()`
		…
		Perform `syscall`
		trap into OS
	Save regs to *kernel stack*	
	Move to kernel mode	
	Jump to trap handler	
Handle trap		
Do syscall work		
return-from-trap		
	Restore regs from *kernel stack*	
	Move to user mode	
	Jump to PC after trap	
		…
		Return from `main()`
		trap (via `exit()`)
Free memory of process		
Remove from proc. list		

Question 2: Regaining Control

The OS wants to switch to another process…No problem – except: *It is not running on the CPU anymore!*

Two possible solutions:

Cooperation
: Provide a special *yield* syscall or simply wait for the next regular one. They happen often.

OS takes control
: Using a timer interrupt (requires hardware), "automatic" return to the OS at regular intervals is possible.

→ Which one is better, what do you think? What are the pros and cons? What if a timer interrupt occurs during another timer interrupt?

Context Switches

A context switch occurs, when the operating system switches from one process to another. It consists of a few steps:

- While switching to *kernel mode*, general purpose registers and SP/PC of the currently running process are saved on the kernel stack
- In kernel mode, they are then saved to the corresponding process structure
- A *scheduling decision* takes place
- The registers of the next process to run are restored from its process structure to kernel stack
- A *return-from-trap* takes place
- The registers are restored from kernel stack while switching to *user mode*

Limited Direct Execution
(Timer Interrupt)

OS@boot (kernel mode)	Hardware
Initialize trap table	
	Store addr. of syscall handler *Store addr. of timer handler*
Start interrupt timer	
	Start timer Interrupt CPU in X ms

OS @run (kernel mode)	Hardware	Process (user mode)
		Process A ...
	Timer interrupt Save regs(A) to *kernel stack (A)* Move to kernel mode Jump to trap handler	

Limited Direct Execution (cont.)
(Timer Interrupt)

OS @run (kernel mode)	**Hardware**	**Process (user mode)**
Handle trap Call **`switch()`**… …save regs(A) to struct(A) …restore regs(B) from struct(B) …switch to *kernel stack (B)* **return-from-trap (into B)**		
	Restore regs(B) from *kernel stack (B)* Move to user mode Jump to PC (B)	
		Process B …

The *xv6* `swtch` Function

Source: [xv6b]

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Creating Processes

The `fork()` and `wait()` Syscalls

On UNIX systems, new processes are *created* using the **`fork()`** syscall. It has an interesting behavior:

- It creates an almost exact copy of the calling process
  - The child gets a *copy* of data, stack and heap; the text section is shared
  - Open file descriptors are duplicated
  - Numerous other properties are inherited – see "`man 2 fork`"
- It *returns twice*, once in the parent process and once in the child process
  - In the parent, the return value is the PID of the child; in the child it is 0 (why?)
  - Which process returns first is *not deterministic*

The **`wait()`** syscall enables the parent process to wait for child termination (and some other state changes – see "`man 2 wait`")

Example for `fork()` and `wait()`

Source: [ost], cpu-api/p2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    sleep(1);
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
    rc, wc, (int) getpid());
  }
  return 0;
}
// Output:
// hello world (pid:22103)
// hello, I am child (pid:22104)
// hello, I am parent of 22104 (wc:22104) (pid:22103)
```

The `exec()` Functions

To start a *different* program, the child process can use one of the **`exec()`** functions (which are front-ends to the **`execve()`** syscall):

- Different options available for parametrization: **`execl()`**, **`execlp()`**, **`execle()`**, **`execv()`**, **`execvp()`** and **`execvpe()`**
- Replace the *current* process by loading a new program
  - Load code and static data
  - Reinitialize heap, stack and other parts of memory
  - Run the program with arguments, environment etc.
- Does *not* return (if no error occurred)
- The **`fork()`** and **`exec()`** pattern allows to *modify the environment* when starting a new program

Example for exec()

Source: [ost], cpu-api/p3.c

```
int main(int argc, char *argv[])
{
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {
    // parent goes down this path (original process)
    int wc = wait(NULL);
    printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
       rc, wc, (int) getpid());
  }
  return 0;
}

// Output:
// hello world (pid:22615)
// hello, I am child (pid:22616)
//  32 123 966 p3.c
// hello, I am parent of 22616 (wc:22616) (pid:22615)
```

Function or Syscall?

You may have noticed, that functions and syscalls seem to look the same so far...

- If it looks the same, how can the system decide what is is?
- Actually, these were all *function calls*
  - A system call is low level: Place arguments in registers, invoke the trap instruction, retrieve returned data...
  - The C library offers functions which perform these steps, as for instance `fork()` and the different variants of `exec()`

Appendix

Bibliography

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

[ost] *GitHub.com, remzi-arpacidusseau/ostep-code: Code from various chapters in OSTEP (http://www.ostep.org)*, `https://github.com/remzi-arpacidusseau/ostep-code`.

[SR13] W. Richard Stevens and Stephen A. Rago, *Advanced programming in the unix environment*, 3 ed., Addison-Wesley professional computing series, Addison-Wesley, 2013.

[xv6a] *GitHub.com, mit-pdos/xv6-public: xv6 OS, proc.h*, `https://github.com/mit-pdos/xv6-public/blob/master/proc.h`.

[xv6b] *GitHub.com, mit-pdos/xv6-public: xv6 OS, swtch.S*, `https://github.com/mit-pdos/xv6-public/blob/master/swtch.S`.