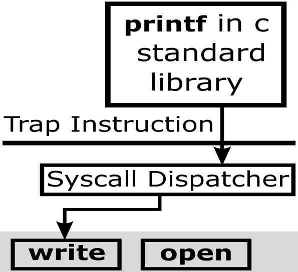
Operating Systems

02. Process

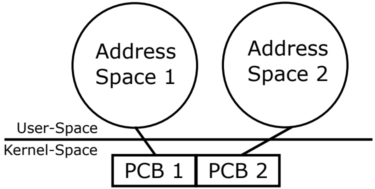
Prof. Dr. Frank Bellosa | WT 2020/2021



Process: Definition

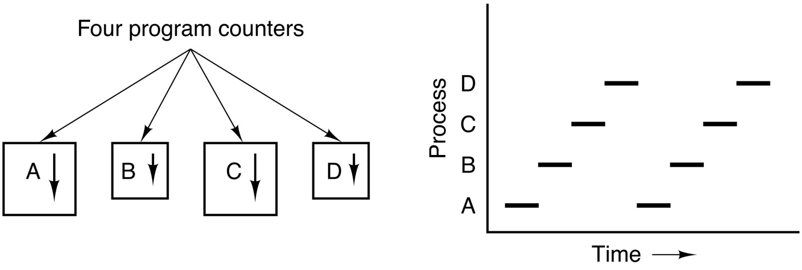
- A process is a program in execution – an “instance” of a program
  → An execution stream in the context of a process state
  - Execution stream: sequence of executed instructions
  - Process state: Everything that the running code can affect or be affected by
    - Registers: general purpose, floating point, status, program counter, stack pointer
    - Address space: all memory locations a program can name
    - I/O state: view of all active I/O activities e.g., open files

    → Each process is associated with a OS data structure called process control block (PCB) that holds all state information
- Multiple processes can coexist in a system (Multiprogramming)



Concurrency vs. Parallelism

- The OS uses both concurrency and parallelism to implement multiprogramming
  (a) Concurrency/Pseudoparallelism: Multiple processes on the same CPU



[TB15]

  (b) Parallelism Processes truly running at the same time with multiple CPUs

In this lecture we will focus on concurrency

Limited Direct Execution

How to Provide Good CPU Performance?

- Direct execution
  - Allow user process to run directly on hardware
  - OS creates process and transfers control to starting point (i.e., main())

- Problems with direct execution:
  - Process can harm each other
  - Could read/write other process data (memory)
  - Process could run forever (slow, buggy, or malicious)
  - Process could do something slow (like I/O)

→ Solution

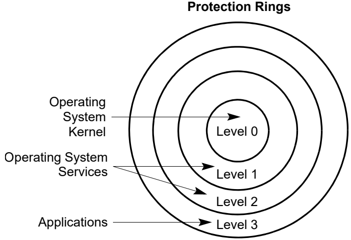
Limited direct execution– OS and hardware maintain some control

Limited Direct Execution

- CPU modes of execution (bit of status)
  - User processes run in user mode (restricted mode)
  - OS runs in kernel mode (not restricted)
    - OS creates process and transfers control to starting point (i.e., main())
    - Change privilege level through system call (trap instruction)
      System calls are function call implemented by OS
- Memory Management Unit
  - User processes have their own separated address space
  - Kernel memory is protected from user mode references
- CPU supports notifications (interrupts) from timers and I/O devices
  - OS can preempt a process
- CPU should not stall too long while waiting for slow I/O
  - OS wants to switch CPU to other process

Central Processing Unit (CPU) - Modes of Execution

- User Mode (x86: “Ring 3” or CPL3)
  - Only non-privileged instructions may be executed
  - Cannot manage hardware in this mode → protection!
- Kernel Mode (x86: “Ring 0” or CPL0)
  - All instructions allowed: Can manage hardware with *privileged instructions*

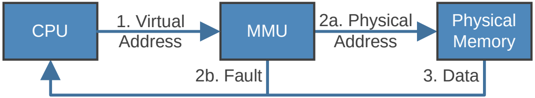


Reference: [int20]

- Modern CPUs have even more modes (Hypervisor, System Management Mode [Fra19])

Virtual Memory Abstraction: Address Spaces

- Every process uses its own virtual addresses (vaddr)
  - Memory-Management Unit (MMU) relocates each load/store to physical memory (pmem)
  - Processes never see physical memory and cannot address it directly



+ MMU can enforce protection (mappings are set up in kernel mode)
  - Processes can only access what they can address (and cannot change mappings)
+ Programs can see more memory than available
  - 80:20 rule: 80% of process memory idle, 20% active working set
  - Can keep working set in RAM and rest on disk (relocate dynamically)

– Need special MMU hardware

A Process's View of the World: Address Space

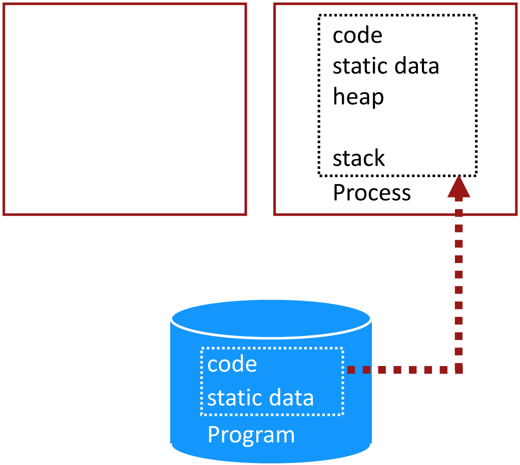
- Code, data, and state need to be organized within processes resulting in an address space layout

- Generally there are three kinds of data
  1. **Fixed size** data items
  2. Data that is naturally **free'd in reverse order of allocation**
  3. Data that is **allocated and free'd dynamically** "at random"

- Compiler and architecture determine e.g., how large an integer is and what instructions are used in the text section (code)

- The loader determines based on an executable file (**`.exe`**, **`.com`**, **`ELF`**) how an executed program is placed in memory

Process Creation



1. Fixed-size Data and Code Segments

- Some data in programs never changes, other data will be written but never grows or shrinks
  - Such memory can be statically allocated when the process is created
- The BSS segment (**B**lock **S**tarted by **S**ymbol, also: .bss or bss)
  - Variables that have not been initialized
  - The executable file typically contains the starting address and size of BSS
  - The entire segment is initially zero
- The data segment
  - Initialized data elements
- The read-only data segment
  - Constant numbers and strings
- The BSS, data, and read-only data segments are sometimes summarized as a single data segment
  - Ultimately the compiler (linker) and operating system (loader) decide where to place which data and how many segments exist

2. Stack Segment

- Some data is naturally free'd in reverse order of allocation
  - **`push( a )`**
  - **`push( b )`**
  - **`pop( b )`**
  - **`pop( a )`**

- Makes memory management very easy (e.g., stack grows downwards)
  - Fixed starting point of segment (not explicitly stored in process)
  - Store bottom of latest allocation **`SP`** (stack pointer), initialized to starting point
  - Allocate new **`a`** byte data structure: **`SP -= a; return SP;`** (push CPU instruction)
  - Free **`a`** byte data structure: **`SP += a;`** (pop CPU instruction)

3. Dynamic Memory Allocation in the Heap Segment

- Some data needs to be allocated and free'd dynamically "at random"
  - E.g., input/output: don't know how large the data will be
  - Don't know how large the text document will get when starting vim

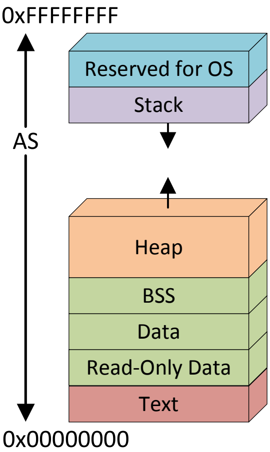
- Generally allocate memory in two tiers:

1. Allocate large chunk of memory (heap segment) from OS
   - Like stack allocation: base address + break pointer (BRK)
   - Process can get more memory from OS or give back memory by setting BRK using a system call (e.g., **`sbrk()`** in Linux)

2. Dynamically partition large chunk into smaller allocations dynamically
   - **`malloc`** and **`free`** commands that can be used in any order
   - This part happens purely in user-space!
     No need to contact kernel at this point!

Typical Process Address Space Layout

OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized data
Data	Initialized data
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

OS Addresses where the kernel is mapped (cannot be accessed by process)

Stack Local variables, function call parameters, return addresses

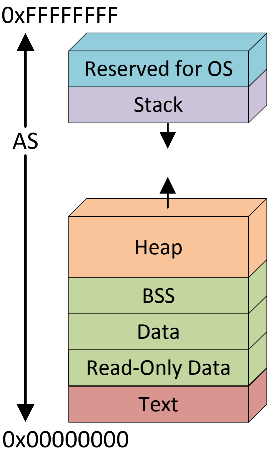
Heap Dynamically allocated data (`malloc`)

BSS Uninitialized data

Data Initialized data

RO-Data Read-only data, strings

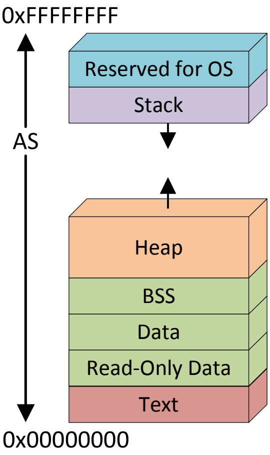
Text Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

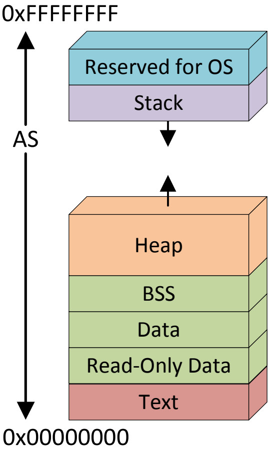
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized data
Data	Initialized data
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

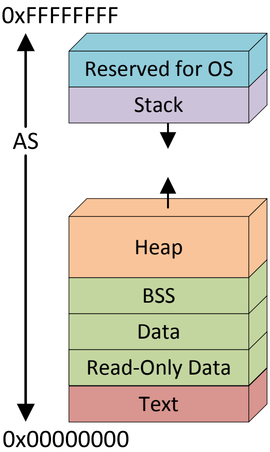
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized data
Data	Initialized data
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

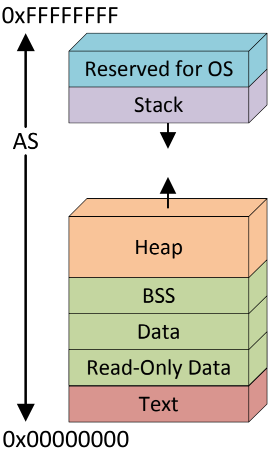
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized data
Data	Initialized data
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

Typical Process Address Space Layout

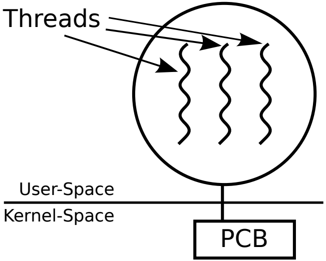
OS	Addresses where the kernel is mapped (cannot be accessed by process)
Stack	Local variables, function call parameters, return addresses
Heap	Dynamically allocated data (`malloc`)
BSS	Uninitialized data
Data	Initialized data
RO-Data	Read-only data, strings
Text	Program, machine code



- Instruction pointer is address in text segment
- Stack pointer is lower-most address of stack segment
- Program break pointer (BRK) is upper-most address of heap segment

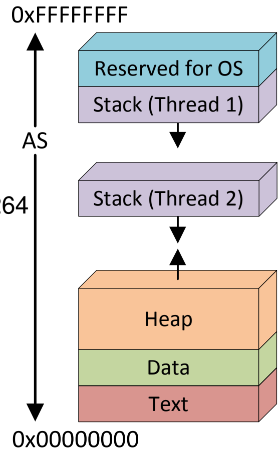
Threads

- Each process consists of ≥ 1 threads representing execution states
  - Instructions pointer register (IP) stores the currently executed instruction
    - An address in the text section
  - Stack pointer (SP) register stores the address of the top of the stack
    - With > 1 threads, there are also multiple stacks!
  - Program status word (PSW) contains flags about the execution history
    - e.g., last calculation was zero → used in following jump instruction
  - And more, e.g., general purpose registers, floating point registers, ...



Processes vs. Threads

- A process is different than a thread
- Thread: Lightweight process (LWP)
  - An execution stream that shares an address space
  - Multiple threads within a single process
- Example:
  - Two processes examining same memory address 0xffe84264 see different values (I.e., different contents)
  - Two threads examining memory address 0xffe84264 see same value (I.e., same contents)



OS Invocation

Invoking the Operating System

- The Operating System Kernel does **not** always run in the background
  - Not even if there are multiple cores/CPUs!

- Three occasions invoke the Kernel and switch to kernel mode

System calls: User Mode process requires higher privileges

Interrupts: CPU external device sends a signal

Exceptions: The CPU signals an unexpected condition

System Call Motivation

- Problem: Want to protect processes from one another
- Idea: Restrict processes by running them in CPU user mode

- Problem: Now processes cannot manage hardware and other protected resources
  - Who can switch between processes?
  - Who decides if the process may open a certain file?
- Idea: The operating system provides services to applications (e.g., hardware management)
  - Application calls the system if service needed (System Call, syscall)
  - OS can check if application is allowed to perform the action that it asks for
  - If application may perform that action and has not exceeded its quota yet, the OS performs the action in kernel mode, on behalf of the application

Types of System Calls

- Process Control
- Memory Management
- File Management
- Device Management
- Communication
- Information Maintenance
- System Management

Examples of Linux System Calls

File management

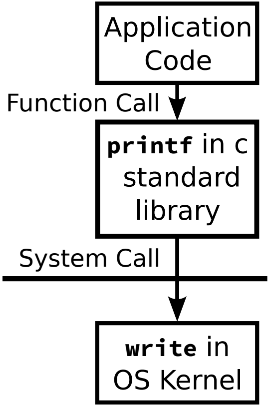
Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file

[TB15]

- In Linux system calls are documented in manual section 2
  - e.g., **`man 2 write`**
- An overview of all syscall is given in **`man 2 syscalls`**

System Calls vs. APIs

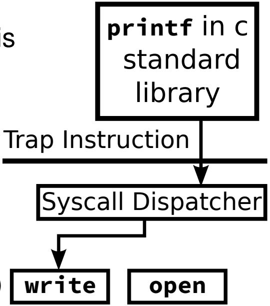
- The syscall interface between applications and OS services provides a limited number of well-defined entry points to the kernel
- Programmers often use syscalls via Application Program Interfaces (APIs)
  - In this example the **`printf`** library call uses the **`write`** system call to output text to the console.



- Most common APIs are
  - Win32 API for Windows
  - POSIX API for virtually all versions of UNIX, Linux, and Mac OS X
  - C API man pages can be found in **`man`** section **`3`** (e.g., **`man 3 printf`**)

System Call Implementation

- Although there are many different system calls, there is only one system call interface (entry point) into the kernel
- The trap instruction is that single entry point
  - The trap instruction switches the CPU to kernel mode and enters the kernel in the same, predefined way for every syscall (e.g., Intel x86: **`sysenter`**, Intel/AMD x86-64: **`syscall`**)
  - The system call dispatcher in the kernel then acts as a multiplexer for all syscalls



- Syscalls are identified by a number which is passed as a parameter
  - The system call table maps system call numbers to kernel functions
  - The dispatcher decides where to jump based on the number and table
  - Programs (e.g., stdlib) have the system call number compiled in. For compatibility: never reuse old numbers in future versions of kernel!

Syscall Parameter Passing (Example)

1-4 Is a library function call. The program pushes parameters for the **`read`** syscall and calls the syscall wrapper from **`unistd.h`**.

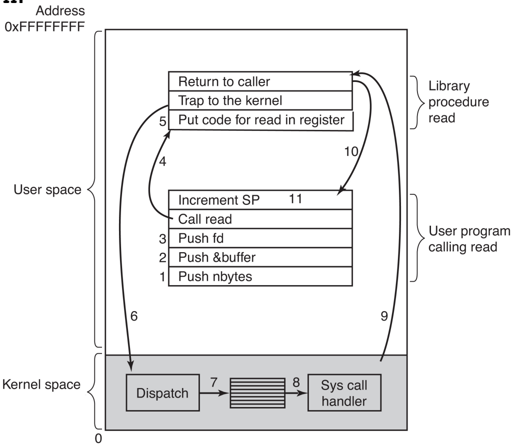
5 Set up syscall number and parameters

- Here, parameters are passed via the stack and are already in the right place
- The system call number is passed via register

6 Caller traps into the kernel

7-8 The dispatcher looks up the syscall number and calls the correct handler

9-11 The kernel returns after finishing the services or in case of an error [TB15]



System Call Handler

- The System Call Handler implements the actual service

1. Saves registers that it taints
2. Reads the parameters that were passed by the caller
3. **Sanitizes/checks the parameters**
4. Checks if the process has permission to perform the requested action
5. Performs the requested service on behalf of the process
6. Returns to the caller with a success or error code

- Checking parameters and permissions is crucial
  - Many bugs in syscall handlers have led to privilege escalation in the past

Interrupts

- Devices use interrupts to signal predefined conditions to the OS
  - The device has an “interrupt line to CPU”
    e.g., device controller informs CPU that it has finished an operation
- The Programmable Interrupt Controller manages interrupts (e.g., x86 APIC)
  - Interrupts can be masked (ignored for now)
  - Masked interrupts are queued and delivered when the interrupt is unmasked
  - The queue has finite length → interrupts can get lost
- Notable examples for interrupts are
  - e.g., *Timer Interrupt* periodically interrupts processes and switches to kernel
    → Can then switch to different process to enforce fairness between processes
  - e.g., *Network Interface Card* interrupts CPU when a packet was received
    → Can deliver the packet to process and free the NIC buffer

Interrupts

- When interrupted, the CPU
  - looks-up the interrupt vector, a table that is pinned in memory and contains the addresses of all service routines (set up by the OS)
  - transfers control to the respective interrupt service routine in the OS that handles the interrupt

- The interrupt service routine must first save the state of the interrupted process
  - Instruction pointer
  - Stack pointer
  - Status word

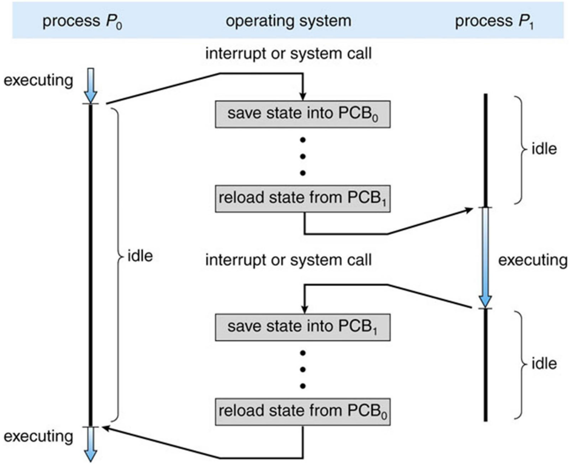
Exceptions

- Sometimes, an unusual condition makes it impossible for the CPU to continue processing
  - What should happen if a process calls `div` with a zero denominator?
  - What should happen if a process tries to write a read-only memory area?
  - What if the process jumps to an invalid opcode?
- On such occasions, an exception is generated within the CPU
  - The CPU immediately stops the process and transfers control to the kernel
  - The kernel can determine the reason for the exception
  - If the kernel can resolve the problem it does so and continues the faulting instruction
  - Otherwise it kills the process
- In addition to the source, there is another distinction between interrupts and exceptions
  - Interrupts can happen in any context
  - Exceptions always occur synchronous to and in the context of a process or the kernel

Voluntary Yielding vs. Preemption

- The kernel is responsible for performing the CPU switch
- The kernel does not always run and cannot dispatch a different process unless it is invoked
  - The kernel can switch at any system call
  - Using cooperative multitasking, the currently running process performs a yield system call to ask the kernel to switch to another process
- The kernel often wants to preempt the currently running process to schedule a different process
  - Preemptive scheduling requires the kernel to be invoked in certain time intervals
  - In general, the kernel uses the timer interrupt as a trigger to make scheduling decisions after every time slice

CPU Switch From Process to Process



[SGG12]

Limited Direct Execution with Timer Interrupts

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of... syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) → proc_t(A) restore regs(B) ← proc_t(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B ...

Process API

Process Creation

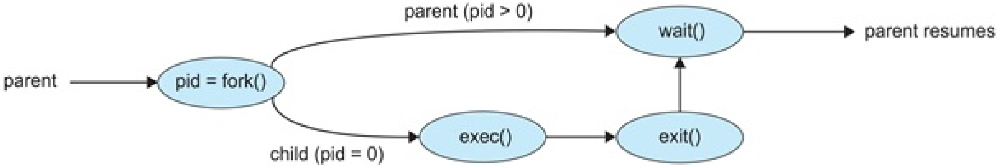
- Four events cause processes to be created
  1. System initialization (booting)
  2. Process creation syscall issued by a running process
  3. User request to create a new process
  4. Initiation of a batch-job

- Those events all actually map to the same two mechanisms
  - The Kernel spawns the initial user space process on boot
    1. Linux: init oder systemd
  - User space processes can spawn further processes (within their quota)
    2. Windows: **`CreateProcess`**, POSIX: **`fork`**[For]
    3. Windows: e.g., click on file
       → explorer.exe calls **`CreateProcess`**
    4. Linux: e.g., cron daemon is started on boot
       → starts batch jobs defined in cron table

POSIX Process Creation using `fork`

- Every process is identified by its process identifier (PID)
- `pid = fork()` duplicates the current process
  - The call returns 0 to the new child
  - It returns the new process PID to the parent
  - → Can continue differently in parent and child process after `fork`
- `exec(name)` replaces own memory based on an executable file
  - `name` specifies the binary executable file
- `exit(status)` terminates own process and returns an exit status
- `pid = waitpid(pid, &status)` wait for termination of a child
  - Pass `pid` of process to wait for as argument
  - `status` points to a data structure that returns information about the process, e.g., the exit status
  - The passed pid is returned on success, otherwise `-1` indicates failure

POSIX Process Creation using `fork`



[SGG12]

Process Environment

- You can pass environment variables when creating a process
- The environment is typically defined by your shell (type **`env`** in Linux)

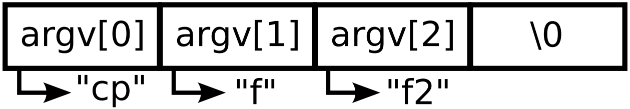
```
$ env
[...]
SHELL=/bin/bash
TERM=xterm-256color
[...]
USER=bellosa
[...]
EDITOR=emacs
```

- Further environment variables are passed with **`execvpe`**

Command Line Arguments

- You can pass arguments to a process at creation
  - **`$ cp f f2`** – execute program cp with arguments “f” and “f2”
  - Flags are arguments given with a special leading character
    - e.g., Windows uses **/** character: try **`copy.exe /?`** in **`cmd.exe`**
    - e.g., Linux and Mac OS use **–** character: try **`cp –r dir1 dir2`** in terminal
    - e.g., Linux and Mac OS also have long options **–**: try **`cp –help`**
  - Clicking a file in Windows or Linux is really just calling the default handler with the filename as the argument
    - In Linux this equates to **`xdg–open <filename>`**

- Arguments are passed as a vector of strings
  - Arguments are specified when using **`execl`** or **`execv`**
  - The flag format is just a convention → all arguments are simply strings



Passing the Argument Vector

- In C, programs begin executing at the **`main`** function

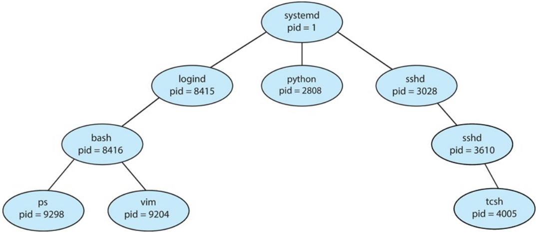
  ```
  int main( int argc, char *argv[], char *envp[] );
  ```

- In principle, the OS calls **`main`** with the arguments given to **`execve`**
  - **`argc`** is the argument count, the number of arguments
  - **`argv`** and **`envp`** are the argument and environment vector pointers
  - If **`execv`** is used, then **`envp = NULL`**

- In C, the main function is handled just like any other function in regard to its stack representation
  - The OS writes the arguments' strings (e.g., "cp", "f", "f2") somewhere in memory (e.g., in the data section)
  - The OS then creates an initial process stack by pushing the argv pointers that contain the memory addresses of those argument strings
  - Finally, the **`IP`** of the process is set to the **`main`** label

POSIX Process Hierarchy

- Parent process creates child processes, which in turn create other processes, forming a process tree



- Parent and children share resources (parts of the AS)
- Parent and children execute concurrently
- Parent waits until children terminate to collect their exit status (with **`waitpid`**) [SGG12]

Daemons

- Some processes are designed to run in the background
  - e.g., a web server
- Those daemons are detached from their parent process after creation
  - This can be done by creating a new session using **`setsid`** in C
  - In bash this can be done with **`disown`**
- Daemons are (re-)attached directly to the root of the process tree (init)
  - init automatically collects their exit status (and ignores it)
- On your Linux machine you can check out the process structure with **`pstree -a`**

Process Termination

Four events cause processes to terminate:

1. Normal exit (voluntary)
   - `return 0;` at end of `main` or `exit(0);`
2. Error exit (voluntary)
   - `return x;` at end of `main`, `exit(x)`, or `abort();` (x != 0)
3. Fatal error (involuntary)
   - OS kills process after exception (e.g., illegal instruction or memory reference)
   - Process exceeds allotted resources (`man ulimit`)
4. Killed by another process (involuntary)
   - Another process sends a signal to kill the process
   - Only with permission (parent process or administrator privileges)
   - e.g., Windows: `TerminateProcess`
     e.g., Linux: `kill(<pid>, -9);` (see `man 7 signal`)

Exit Status

- Processes return their exit status in form of an integer on voluntary exit
  - In Linux only the last 8 Bits are significant, regardless of the integer's size
- The process resources cannot be completely free'd after it terminates
  - A Zombie or process stub, that can deliver the exit status remains until it is collected via **`waitpid`**. Only then can the PID be free'd and all resources deallocated
- Children that keep running after their parent exits are called orphans
  - Today, init generally adopts orphans – they keep running.
    Init collects and ignores the exit status on exit
  - Some systems perform a cascading termination → The OS kills all children when a parent exits
- On involuntary exits of children
  - Bits 0-6 contain the signal number that killed the process (0 on normal exit)
  - Bit 7 is set if the process was killed by a signal
  - Bits 8-15 are 0 if killed by signal (exit status on normal exit)

References I

[ADAD18] R.H. Arpaci-Dusseau and A.C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2018.

retrieved from http://pages.cs.wisc.edu/~remzi/OSTEP/.

[For]

[Fra19] Jessie Frazelle. Open source firmware. *Commun. ACM*, 62(10):34–38, September 2019.

[int20] *Four-Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals*, 2020. retrieved from https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html#three-volume.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

References II

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 4th edition, 2015.