Karlsruhe Institute of Technology

Betriebssysteme

04. Process API

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP

Where we ended last lecture

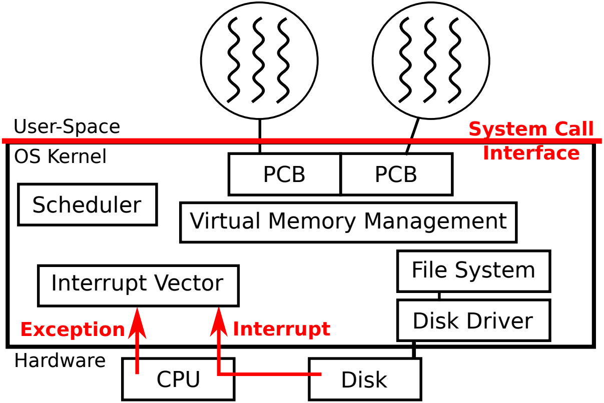
- The OS provides abstractions for and protection between application
  - Processes run without privileges in user-space
  - Kernel governs resources and runs in kernel-space
  - The distinction between kernel and user-space is made in the CPU
  - If if a process executes a privileged instruction, the CPU calls the kernel instead

- Address space: all memory the process can name
  - Stack: Local variables, function call parameters, return addresses
  - Heap: Dynamically allocated data (**`malloc`**)
  - Data: Global variables, strings
  - Text: Program, machine code

Where we ended the lecture before that

- OS does **not** always run in the background! Invoke the OS with:

system call	application calls OS to request a service
interrupt	device calls OS to signal an event
exception	CPU calls OS to signal an error/special condition



Execution Model

Assembler

- The OS interacts directly with compiled programs
  - Switch between processes and threads → save and restore state
  - Deal with and pass on signals and exceptions
  - Receive requests from applications
- To understand some OS principles you need to know basic CPU and hardware details
  - We assume that you have already studied some assembler in another class
  - We use the following simplified instruction names in this lecture for clarity
- Data Movement

  mov Copy data referenced by second operand to location referenced by first operand

x86 Arithmetic Commands

add/sub Different forms (memory/registers) add, subtract, multiply, or divide two integer operands storing result in first operand

inc/dec Increment (add one) or decrement (subtract one) from a register or memory location

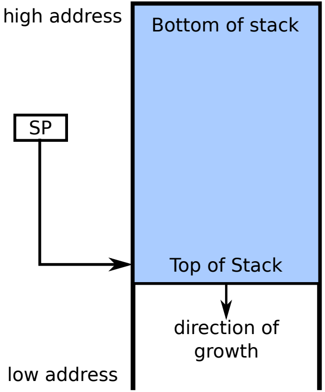
shl/shr Shift first operand left/right by a number of bits given by second operand

and/or/xor Calculate bitwise and/or/exclusive or of two operands storing the result in first operand

not Logically negate operand

x86 Stack

- Stack Pointer register (SP) holds the address of the top of the stack
  - Stacks grows downwards
  - SP points at last allocated word of the stack ("pre-decrement stack pointer")



- Push makes room for values on the stack by decrementing the SP and placing the new element in the newly allocated area
- Pop cleans up values from the stack by incrementing the SP (the removed data is not overwritten)
- Base Pointer (BP) register (a.k.a. frame pointer) can be used to organize larger chunks of the stack called stack frames

Basic x86 jump/branch/call Commands

jmp Continue execution at address given in operand.

j$condition Jump conditional depending on PSW content.
If condition is true jump, otherwise just go to next instruction.
$condition examples: je (jump equal), jz (jump zero).

call Used to jump to a function (subroutine). Push the current code location onto the stack and perform an unconditional jump to the function address.

return Used to return from a function. Jumps to the return address on stack.

Application Binary Interface

- The Application Binary Interface (ABI) standardizes binary interface between programs/modules/the OS
  - Specifies executable/object file layout, calling convention, alignment rules
  - Example: System V AMD64 ABI used in Linux, BSD and Mac OS X
- Calling conventions standardize the exact way function calls are implemented to achieve interoperability between compilers
- C (historically) defined such conventions under the name **`cdecl`**

x86 Calling Conventions

- When a function is called, the caller
  1. Saves the state of the local scope
  2. Sets up parameters where the subroutine can find them
  3. Transfers control flow

- The called function then:
  4. Sets up a new local scope (local variables)
  5. Performs its duty
  6. Puts the return value where caller can find it
  7. Jumps back to calling function (IP)

- Functions can call other functions
  - This is done at step 5

Example: cdecl with caller clean-up

- We will call the general purpose registers accumulator (A), base (B), counter (C), data (D), stack pointer (SP), and base pointer (BP)
  - Depending on the architecture they might have an 'X', 'L' or 'H' suffix and different prefixes such as 'E' or 'R' in other publications
- In cdecl A, C, and D are caller-saved, other registers are callee-saved (e.g., floating point)
- Function arguments are passed via the stack
  - Arguments are pushed in reverse order
  - → Variable parameter number possible via format string in first argument
- Return address is saved on stack
- Return value is
  - passed via stack
  - or via A+D registers

Example: cdecl – C functions

- **`caller`** calls function callee and passes two arguments, 23 and 42.
- **`callee`** adds the arguments and returns the result.
- **`caller`** saves the result in **`res`**.

```
#include <inttypes.h>

__attribute__((__cdecl__))
uint32_t callee( uint32_t a, uint32_t b )
{
    uint32_t c = a + b;
    return c;
}

void caller()
{
    uint32_t res = callee( 23, 42 );
}
```

- You can look at the assembly code with:

`$ gcc -m32 -mpreferred-stack-boundary=2 -g -c source.c -o out.o`

`$ objdump -Sd out.o`

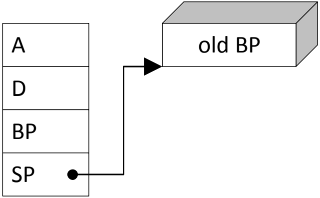
Example: cdecl – Caller (simplified)

```
void caller()
{
   uint32_t res = callee( 23, 42 );
}
```

```
push BP            ; save old stack frame
mov  BP     SP     ; initialize new frame
sub  SP     12     ; reserve memory for
                   ; local vars and arguments

mov  SP+4   42     ; put arguments on stack
mov  SP     23     ; (reverse order!)

call callee        ; save IP and jump to callee
```



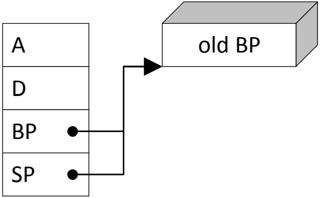
Example: cdecl – Caller (simplified)

```
void caller()
{
   uint32_t res = callee( 23, 42 );
}
```

```
push  BP          ; save old stack frame
mov   BP     SP   ; initialize new frame
sub   SP     12   ; reserve memory for
                  ; local vars and arguments

mov   SP+4   42   ; put arguments on stack
mov   SP     23   ; (reverse order!)

call  callee      ; save IP and jump to callee
```



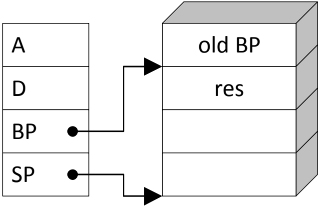
Example: cdecl – Caller (simplified)

```
void caller()
{
   uint32_t res = callee( 23, 42 );
}
```

```
push BP            ; save old stack frame
mov  BP    SP      ; initialize new frame
sub  SP    12      ; reserve memory for
                   ; local vars and arguments

mov  SP+4  42      ; put arguments on stack
mov  SP    23      ; (reverse order!)

call callee        ; save IP and jump to callee
```



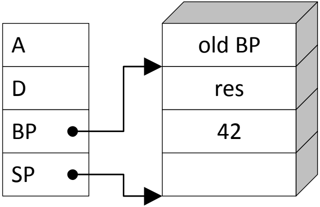
Example: cdecl – Caller (simplified)

```
void caller()
{
   uint32_t res = callee( 23, 42 );
}
```

```
push BP              ; save old stack frame
mov  BP    SP        ; initialize new frame
sub  SP    12        ; reserve memory for
                     ; local vars and arguments

mov  SP+4  42        ; put arguments on stack
mov  SP    23        ; (reverse order!)

call callee          ; save IP and jump to callee
```



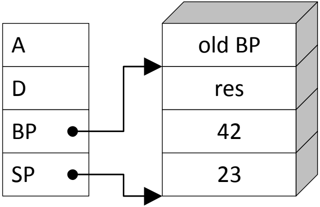
Example: cdecl – Caller (simplified)

```
void caller()
{
    uint32_t res = callee( 23, 42 );
}
```

```
push BP                ; save old stack frame
mov  BP    SP          ; initialize new frame
sub  SP    12          ; reserve memory for
                       ; local vars and arguments

mov  SP+4  42          ; put arguments on stack
mov  SP    23          ; (reverse order!)

call callee            ; save IP and jump to callee
```



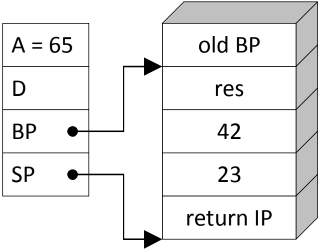
Example: cdecl – Caller (simplified)

```
void caller()
{
   uint32_t res = callee( 23, 42 );
}
```

```
push BP            ; save old stack frame
mov  BP    SP      ; initialize new frame
sub  SP    12      ; reserve memory for
                   ; local vars and arguments

mov  SP+4  42      ; put arguments on stack
mov  SP    23      ; (reverse order!)

call callee        ; save IP and jump to callee
```



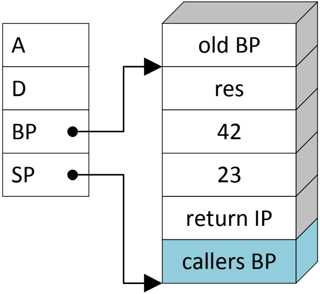
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                  ; function label
 push  BP                ; save frame pointer
 mov   BP    SP          ; create new frame
 sub   SP    4           ; make room for c

 mov   A     BP+12       ; fetch 42 into A
 mov   D     BP+ 8       ; fetch 23 into D
 add   A     D           ; A = A + D
 mov   BP-4  A           ; put result into variable c

 mov   A     BP-4        ; put return value into A
 leave                   ; restore old stack frame
 ret                     ; jump back to caller
```



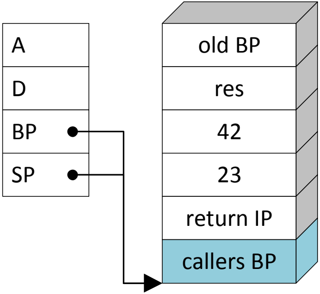
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                  ; function label
 push BP                 ; save frame pointer
 mov  BP    SP           ; create new frame
 sub  SP    4            ; make room for c

 mov  A     BP+12        ; fetch 42 into A
 mov  D     BP+ 8        ; fetch 23 into D
 add  A     D            ; A = A + D
 mov  BP-4  A            ; put result into variable c

 mov  A     BP-4         ; put return value into A
 leave                   ; restore old stack frame
 ret                     ; jump back to caller
```



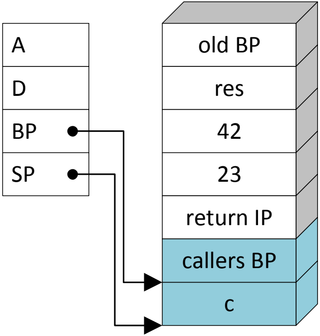
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                 ; function label
 push BP                ; save frame pointer
 mov  BP    SP          ; create new frame
 sub  SP    4           ; make room for c

 mov  A     BP+12       ; fetch 42 into A
 mov  D     BP+ 8       ; fetch 23 into D
 add  A     D           ; A = A + D
 mov  BP-4  A           ; put result into variable c

 mov  A     BP-4        ; put return value into A
 leave                  ; restore old stack frame
 ret                    ; jump back to caller
```



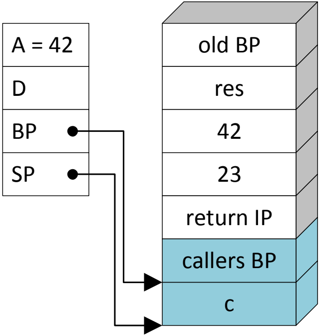
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                     ; function label
 push  BP                   ; save frame pointer
 mov   BP    SP             ; create new frame
 sub   SP    4              ; make room for c

 mov   A     BP+12          ; fetch 42 into A
 mov   D     BP+ 8          ; fetch 23 into D
 add   A     D              ; A = A + D
 mov   BP-4  A              ; put result into variable c

 mov   A     BP-4           ; put return value into A
 leave                      ; restore old stack frame
 ret                        ; jump back to caller
```



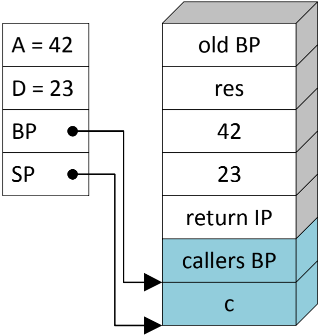
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                     ; function label
 push BP                    ; save frame pointer
 mov  BP   SP               ; create new frame
 sub  SP   4                ; make room for c

 mov  A    BP+12            ; fetch 42 into A
 mov  D    BP+ 8            ; fetch 23 into D
 add  A    D                ; A = A + D
 mov  BP-4 A                ; put result into variable c

 mov  A    BP-4             ; put return value into A
 leave                      ; restore old stack frame
 ret                        ; jump back to caller
```



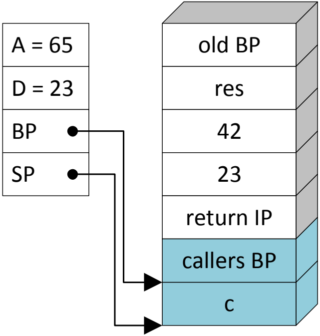
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                 ; function label
 push BP                ; save frame pointer
 mov  BP   SP           ; create new frame
 sub  SP   4            ; make room for c

 mov  A    BP+12        ; fetch 42 into A
 mov  D    BP+ 8        ; fetch 23 into D
 add  A    D            ; A = A + D
 mov  BP-4 A            ; put result into variable c

 mov  A    BP-4         ; put return value into A
 leave                  ; restore old stack frame
 ret                    ; jump back to caller
```



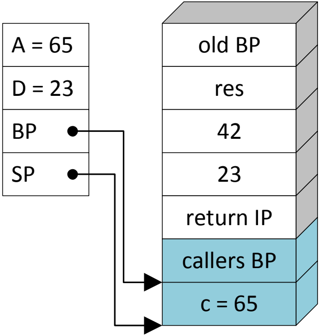
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                  ; function label
 push  BP                ; save frame pointer
 mov   BP    SP          ; create new frame
 sub   SP    4           ; make room for c

 mov   A     BP+12       ; fetch 42 into A
 mov   D     BP+ 8       ; fetch 23 into D
 add   A     D           ; A = A + D
 mov   BP-4  A           ; put result into variable c

 mov   A     BP-4        ; put return value into A
 leave                   ; restore old stack frame
 ret                     ; jump back to caller
```



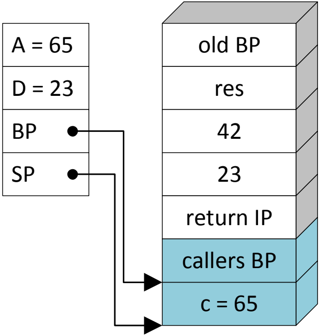
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                  ; function label
 push BP                 ; save frame pointer
 mov  BP   SP            ; create new frame
 sub  SP   4             ; make room for c

 mov  A    BP+12         ; fetch 42 into A
 mov  D    BP+ 8         ; fetch 23 into D
 add  A    D             ; A = A + D
 mov  BP-4 A             ; put result into variable c

 mov  A    BP-4          ; put return value into A
 leave                   ; restore old stack frame
 ret                     ; jump back to caller
```



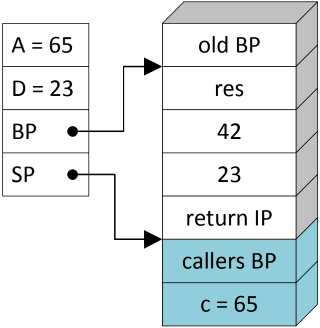
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                   ; function label
 push BP                  ; save frame pointer
 mov  BP   SP             ; create new frame
 sub  SP   4              ; make room for c

 mov  A    BP+12          ; fetch 42 into A
 mov  D    BP+ 8          ; fetch 23 into D
 add  A    D              ; A = A + D
 mov  BP-4 A              ; put result into variable c

 mov  A    BP-4           ; put return value into A
 leave                    ; restore old stack frame
 ret                      ; jump back to caller
```



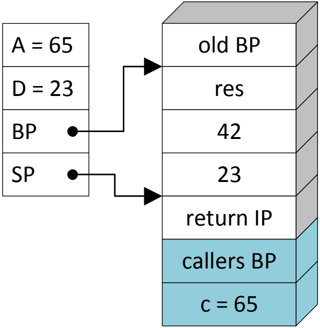
Example: cdecl – Callee (simplified)

```
uint32_t callee( uint32_t a, uint32_t b )
{
   uint32_t c = a + b;
   return c;
}
```

```
callee:                 ; function label
 push  BP               ; save frame pointer
 mov   BP    SP         ; create new frame
 sub   SP    4          ; make room for c

 mov   A     BP+12      ; fetch 42 into A
 mov   D     BP+ 8      ; fetch 23 into D
 add   A     D          ; A = A + D
 mov   BP−4  A          ; put result into variable c

 mov   A     BP−4       ; put return value into A
 leave                  ; restore old stack frame
 ret                    ; jump back to caller
```



Passing Parameters to the System

Parameter Passing and Return

- The system call number must be passed to the kernel along with other parameters which are specific to the called service
- There are different places in which parameters can be transferred
  - A limited number of parameters can be passed via CPU registers ($\sim$ 6)
  - More parameters or data-types such as strings are passed via main memory (heap or stack)
  - All parameters can also be passed via stack or heap
    → the ABI specifies how to pass parameters
- A return code needs to be returned to the application
  - Negative numbers usually used as error codes
  - Positive number and 0 indicate success
- Return codes are usually returned via the A+D registers

Parameter Passing Example

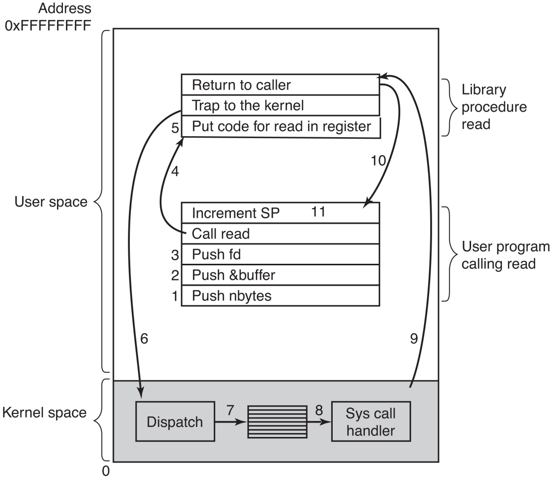
1-4 Is a library function call using cdecl. The program pushes parameters for the **`read`** syscall and calls the syscall wrapper from **`unistd.h`**.

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



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

Process API

Process Creation

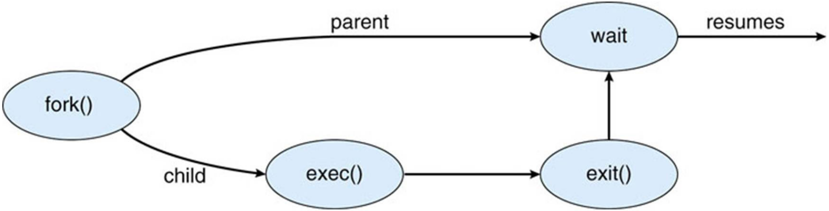
- Four events cause processes to be created
  1. System initialization (booting)
  2. Process creation syscall issued by a running process
  3. User request to create a new process
  4. Initiation of a batch-job

- Those events all actually map to the same two mechanisms
  - The Kernel spawns the initial user space process on boot
    1. Linux: init
  - User space processes can spawn further processes (within their quota)
    2. Windows: `CreateProcess`, POSIX: `fork`
    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`



Process Environment

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

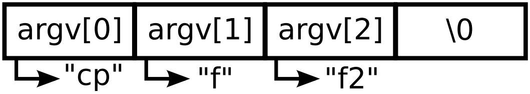
```
$ env
[...]
SHELL=/bin/bash
TERM=xterm-256color
[...]
USER=bellosa
[...]
EDITOR=emacs
```

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

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 **execv** or **execve**
  - The flag format is just a convention → all arguments are simply strings



Passing the Argument Vector

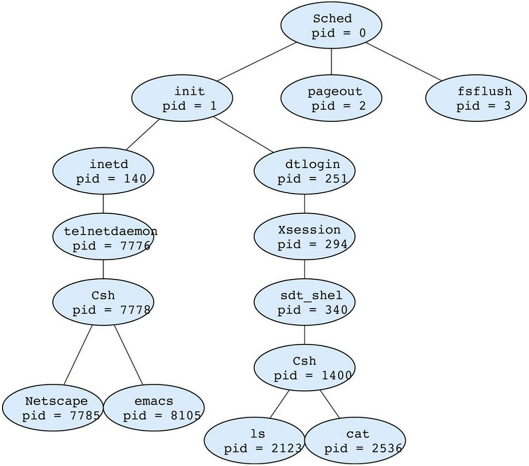
- In C, programs begin executing at the **`main`** function

```
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`**
  - If **`exec`** is used, then **`argc = 0; argv = NULL; 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`**)

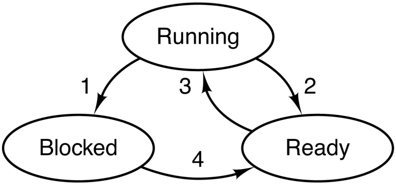


Daemons

- 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 States

- Sometimes processes wait for events or other processes
  - Processes may block (do nothing but wait)
  - This usually happens on system calls
  - OS does not run the process until the event happens
  - e.g., input from keyboard, network packets, child to return, ...



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

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)

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition: Pages 38–50
- Stallings, “Operating Systems – Internals and Design Principles”, 6th Edition: Pages 50–104
- Silberschatz, Galvin, Gagne, “Operating System Concepts”, 8th Edition: Pages 55–66 and 110–116
- Matz, Hubička, Jaeger, Mitchell, “System V Application Binary Interface – AMD64 Architecture Processor Supplement”