

Operating Systems

Introduction, Processes, Threads

Daniel Gruss

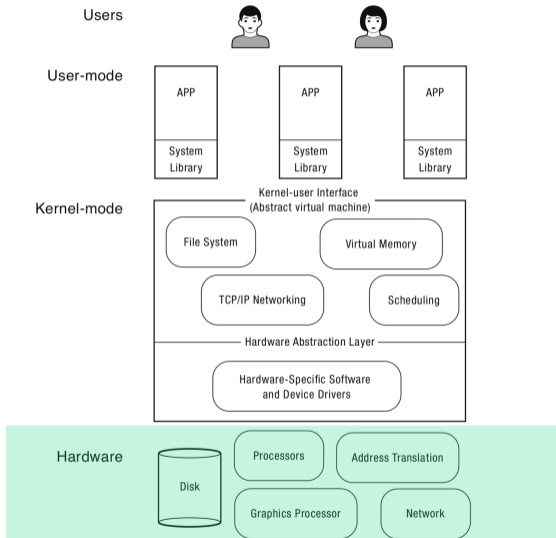
2023-10-08



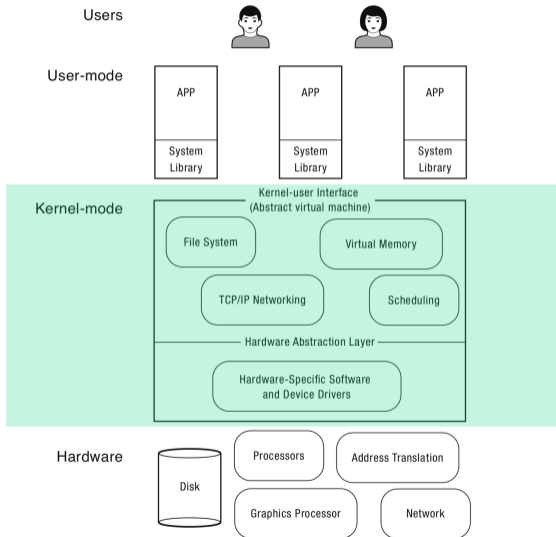
1. Basics
2. Process and Thread Fundamentals
3. Context Switches
4. Process and Thread Organization

Basics

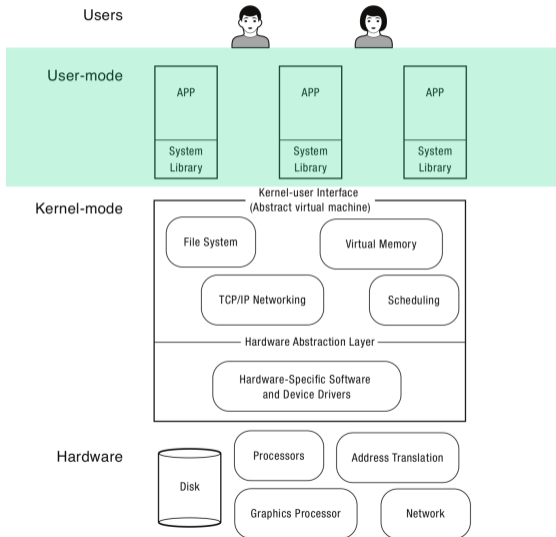
What is an Operating System



What is an Operating System



What is an Operating System



- Run on all sorts of devices:
 - Servers, Desktops, Notebooks
 - Tablets, Smartphones
 - Routers, Switches, Displays
 - Door Locks, Washing Machines, Toasters
 - Cars, Airplanes
 -
- We focus on general purpose operating systems

- Referee 
- Illusionist 
- Glue 

- OS challenges are not unique - apply to many different computing domains
- many complex software systems
 - have multiple users
 - run programs written by third-party developers
 - need to coordinate simultaneous activities

Challenges:

- resource allocation
- fault isolation
- communication
- abstraction
- how to provide a set of common services



Design Criteria for Operating Systems

- Reliability and Availability
- Security
- Portability
- Performance
- Adoption

The first computers were so called “mainframes” that had no operating systems.



- first *collection of compatible utility programs (Multics)*
 - *assemblers, compilers, debugging tools*
 - *standard routines for input and output*
 - *buffers to “spool” printer and tape output*
 - *utilities designed to load sequence (or “batch”) of programs into memory*
 - *automate some of the reconfiguration performed by human operators*

- Multics never gained critical mass in the market place
- Ken Thompson and Dennis Ritchie started working on an OS for microcomputers: UNIX
- by programmers - for programmers
- originally in assembly language
- rewritten in C
- portable operating system!

Phase	Idea
Open shop	operating systems
Batch processing	tape batching, first-in/first -out scheduling
Multiprogramming	processor multiplexing, atomic operations, demand paging, I/O spooling, priority scheduling, remote job entry
Timesharing	simultaneous user interactions, on-line file systems
Concurrent programming	hierarchical systems, extensible kernels, parallel programming
Personal Computing	graphical user interface
Distributed Systems	remote servers

1968: First devices named “personal computer” (actually a calculator)



1973: Xerox Alto, first computer with mouse, desktop, and GUI



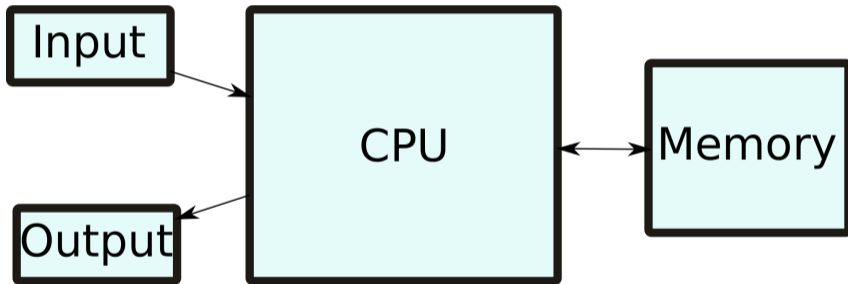
- Different requirements: only one user
- CP/M, DOS, Apple-DOS
- Windows
- OS-2, Windows-XP, OS-X, Linux....

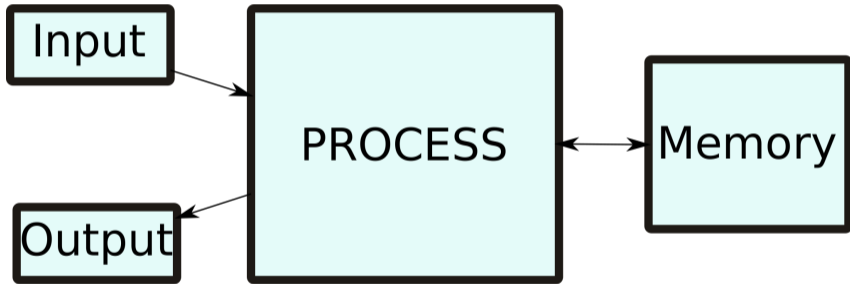
Process and Thread Fundamentals

- A program: a binary file containing code and data
 - actions: write, compile, install, load
 - resources: file
- A thread: an execution context
 - actions: run, interrupt, stop
 - resources: CPU time, stack, registers
- A process: a container for threads and memory contents of a program
 - actions: create, start, terminate
 - resources: threads, memory, program

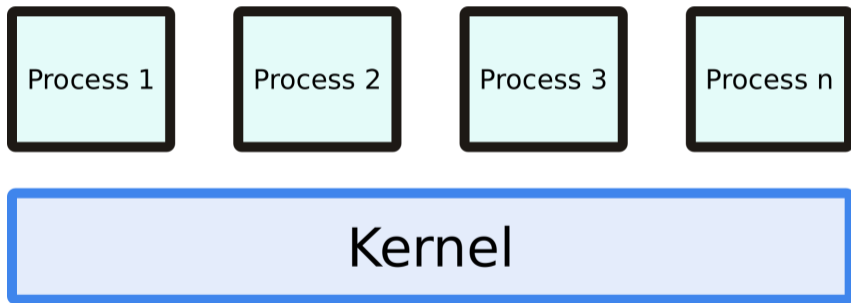
- Process: abstraction of a computer
- File: abstraction of a disk or a device
- Socket: abstraction of a network connection
- Window: abstraction of a display

→ Abstractions hide many details but provide the required capabilities



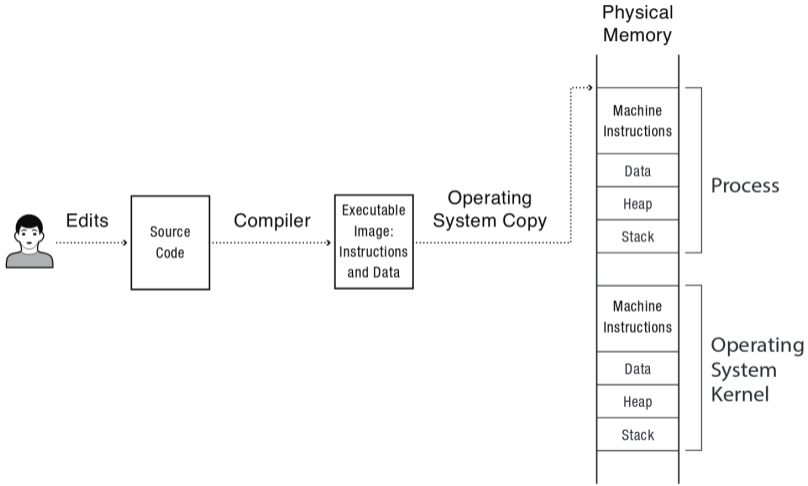


Implemented by the kernel



- We have “one hardware”
- We have many “processes”
- How do we solve this?

The Process Abstraction



- Once a program is loaded in memory, OS can start it(s first thread) by
 - setting up a stack and setting the stack pointer and
 - setting the instruction pointer (of the first thread) to the programs first instruction
- Process is an instance of a program
- Kernel must organize running code of multiple processes
- Must be able to switch from one process to another
- OS keeps a list of process data structure (aka the “PCB”)

Process List (aka PCB)

Process list stores

- where program is loaded in memory
- where image is on disk
- which user asked to execute
- what privileges the process has
- etc.
- Process ID
- User ID
- Process status
- Scheduling information
- I/O resources

Process can have multiple threads

- same program code and data
- own stack
- own registers (including instruction pointer)

Process Protection Mechanisms

Challenges:

- Threads of a process run code
- What code?
- Do we trust that code?
- Maybe buggy? Malicious?
- We want to give the program restricted privileges
- How can we do that?

- Most instructions cannot do any harm
- Some instructions can

```
asm("cli");
```

```
asm("hlt");
```

Examples for Privileged Instructions (Intel)

- LGDT: Load GDT register
- LLDT: Load LDT register
- LTR: Load task register
- LIDT: Load IDT register
- MOV (control registers): Load and store control registers
- LMSW: Load machine status word
- CLTS: Clear task-switched flag in register CR0
- MOV (debug registers): Load and store debug registers
- INVD: Invalidate cache, without writeback
- WBINVD: Invalidate cache, with writeback
- INVLPG: Invalidate TLB entry
- HLT: Halt processor
- RDMSR: Read Model-Specific Registers

- User-mode: limited privileges
- Kernel-mode: complete privileges

Recall: DPL defined in segment descriptor

- User-mode: $DPL = 3$
- Kernel-mode: $DPL = 0$

→ hardware-assisted control mechanisms

Kernel Mode:

- OS runs in kernel mode
- Full privileges for hardware accesses
- Read/write to any memory
- Access to any I/O-device

User Mode:

- User programs run in user mode
- Limited privileges
- Some instructions and memory regions are not accessible
- If tried anyway: exception is raised by the CPU.

- mode stored in EFLAGS register
- segment descriptors
- paging structures
- ...

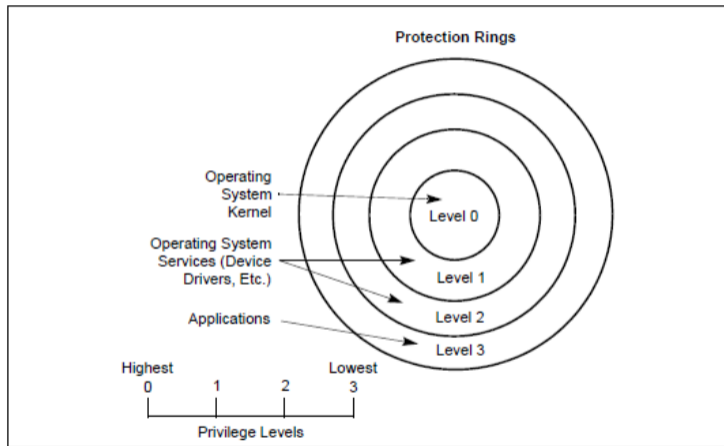


Figure 6-3. Protection Rings

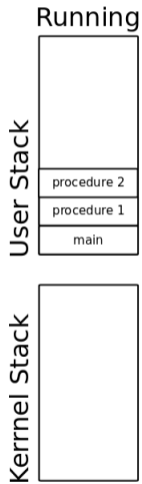
How to change from ring to ring ...

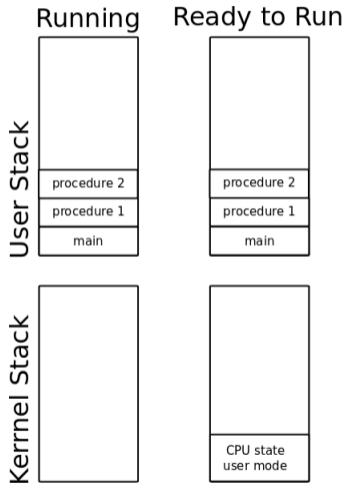
- change from kernel mode (lower level ring) to user mode (higher level ring) not a problem
- change from ring 0 to ring 3 not a problem
- change from ring 0 to ring 3 through special return instruction (`iret`)
- change from user mode (higher level ring) to kernel mode (lower level ring) must be a **controlled procedure**
- change from ring 3 to ring 0 through **controlled procedure**
- change from ring 3 to ring 0 through **int 0x80, sysenter, or syscall**

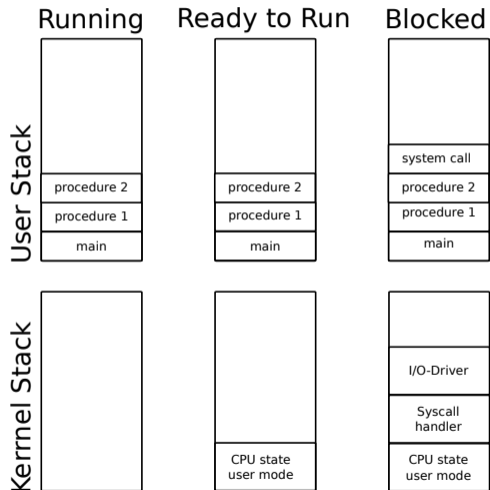
→ Otherwise there would be no protection

- either generated by the software (e.g. syscall)
- or by the hardware
 - timer
 - I/O-devices
 - exceptions (divide-by-zero, page fault, etc.)

- Interrupts switch stack to a kernel stack
- Why?
 - Security and stability
 - Who knows where the users SP points
 - Maybe SP points to illegal address
 - Would raises an page fault exception (in the kernel)
 - Some register values are pushed to stack by the CPU during a context switch
- How many stacks do we actually need?
- Do we need multiple stacks for the kernel?







Context Switches

- one CPU / core: one active thread at any point in time
- how to switch between threads?
- how do we let a CPU / core execute a different function?
- change the instruction pointer? how?

Changing the instruction pointer

```
asm("jmp %[other_thread_function]"  
    :  
    : [other_thread_function]"r"(other_thread_function));
```

does this work? Yes, but ...

- what if the thread is in another process?
- scheduling thread slices: how do we later restore the state we came from?
- what if we're coming from kernelspace?

- Caused only by an interrupt → privilege level change
 - CPU pushes to stack: `ss`, `rsp`, `rflags`, `cs`, `rip`
- Store register values (→ next slide)
- Old thread executes `Scheduler` (code to switch to a new thread)
- Context switch to a new thread

1. Push all CPU register values on the stack
 - No modification of instruction pointer / stack pointer:
 - `rip` and `rsp` were already pushed to the stack by CPU
2. Pop all CPU register values into a struct
3. Set `currentThreadInfo`, etc. to kernel thread

currentThreadRegisters

```
struct ArchThreadRegisters
```

```
{  
    uint64  rip;           // 0  
    uint64  cs;           // 8  
    uint64  rflags;      // 16  
    uint64  rax;         // 24  
    uint64  rcx;         // 32  
    uint64  rdx;         // 40  
    uint64  rbx;         // 48  
    uint64  rsp;         // 56  
    uint64  rbp;         // 64  
    uint64  rsi;         // 72  
    uint64  rdi;         // 80  
    uint64  r8;          // 88  
    uint64  r9;          // 96  
    uint64  r10;         // 104  
    uint64  r11;         // 112  
    uint64  r12;         // 120  
    uint64  r13;         // 128  
    uint64  r14;         // 136  
    uint64  r15;         // 144  
    uint64  ds;          // 152  
    uint64  es;          // 160  
    uint64  fs;          // 168  
    uint64  gs;          // 176  
    uint64  ss;          // 184  
    uint64  dpl;         // 192  
    uint64  rsp0;        // 200  
    uint64  ss0;         // 208  
    uint64  cr3;         // 216  
    uint32  fpu[28];     // 224  
};
```

1. “Restore” CPU register values
 - 1.1 `iretq` (interrupt return) expects `ss`, `rsp`, `rflags`, `cs`, `rip` on the stack
 - 1.2 `iretq` pops values from stack into the registers
2. Instruction pointer has a new value, execution continues there

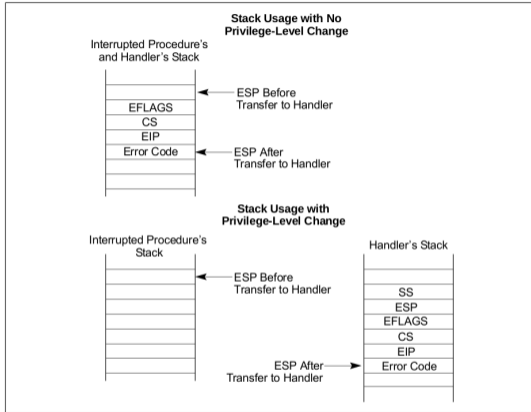


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Looks identical for 64 bits

Act as if:

- Thread was running already
- We are returning from an interrupt

1. Push stored register values to stack (modifies registers)
2. “Restore” CPU register values as before
3. Instruction pointer has a new value, execution continues there

Interrupts:

- clock
- device
- system call (`syscall` / `sysenter` / `int 0x80`)
- cpu fault (trap / interrupt)
 - executing privileged instruction
 - divide by 0
 - integer overflow
 - bad memory access

Process and Thread Organization

- **Program:** a binary file containing code and data
 - a mold for a process
- **Thread:** an execution context
 - a sequence of instructions
 - if part of a process: restricted to the boundaries of a process
- **Process:** a container for threads and memory contents of a program
 - an instance of a program
 - restricted to its own boundaries and rights

A process is a container.

- Process ID
- Filename
- Program file (Loader)
- (Open) file descriptors
- Address space (ArchMemory, CR3 register)
- Accounting
- Threads
- Child processes?

A thread is a unit for execution.

- Thread ID
- Thread state (Running, Sleeping, ...)
- A set of register values (defining the state of the execution of the userspace thread function)
 - Not all registers are different
 - Some register values are process-specific and not thread-specific (e.g. CR3)
- A user stack
- A kernel stack (for syscalls)
- A second set of register values for the kernel (for syscalls)

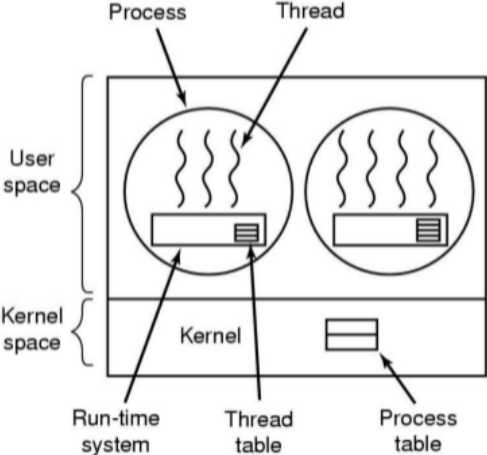
Load program, create process, ...

- 1 initial thread
- executes the `main()`-function
- it's not a "main"-thread
- process may start further threads if required (how?)

- “There is no such thing as a thread” at the CPU-level
- As illustrated before: works by creative and clever usage of interrupts
- Threads can be implemented with and without support of the operating system
 - Pure Userspace Threading: lightweight, but many drawbacks
- Threads can be implemented with and without support of the CPU (`int 0x80` vs. `sysenter/syscall`)

- Kernel has no concept of threads and no idea they might exist (that's how it started)
- Implement threads in userspace as library
- can be implemented in all operating systems

Userspace Threads



- process manages threads
 - user-mode-runtime-system in libc!
- function that might block the thread
 - call method in libc to check: thread going to block?
 - YES: save registers in thread table
 - choose other thread ready to run
 - load chosen the thread's registers from thread table
 - change stack pointer and instruction pointer (this time `jmp`)

- Advantages
 - no system calls for thread handling
 - thread-switches are very fast
 - no change of memory configuration when switching threads
 - can use specialized scheduling

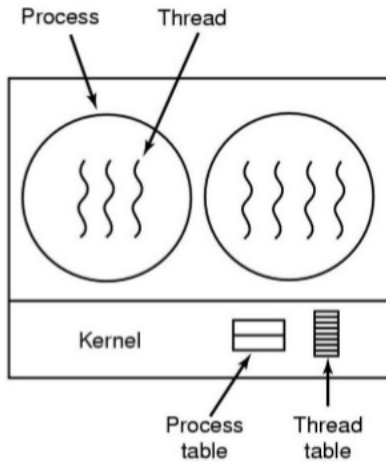
- threads are not allowed to make direct syscalls since they might block
- ... one could make system-calls non-blocking
- ... but since this should work with unchanged (and unaware) OSs...
- sometimes you can find out if a syscall would block
 - e.g. `select-Systemcall`
 - before `read` is called: call `select`
 - should read block: switch threads and check again later
 - not very efficient and elegant
- Sometimes not
 - Page faults
 - if page not in memory, process will block
 - if thread has an endless loop and does not free CPU...

Two and a half options:

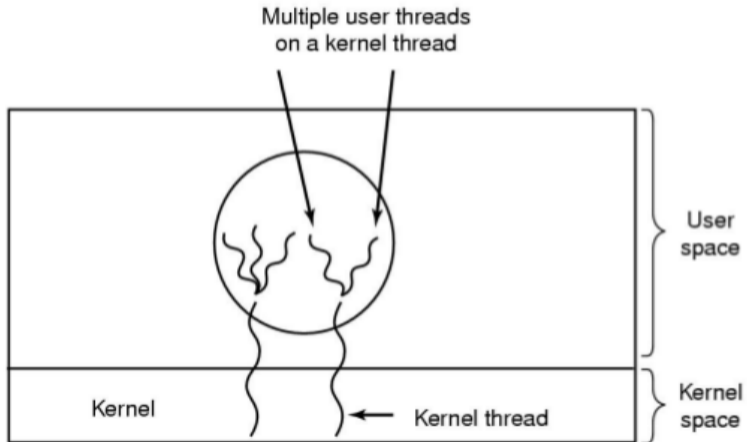
- Userspace
- Kernelspace
- Mixed

- No runtime system needed
 - less code the user can break
- thread management in kernel
 - more or less as in userspace
 - but: kernel programmers *by definition* only write safe code
- thread creation and management via syscall
 - takes longer than before
 - thread-recycling

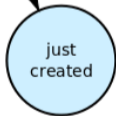
Kernel mode threads

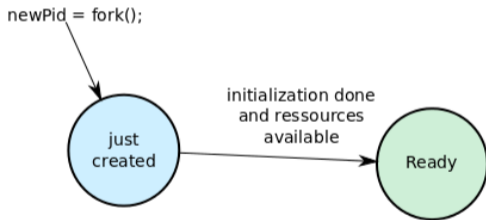


Hybrid solution

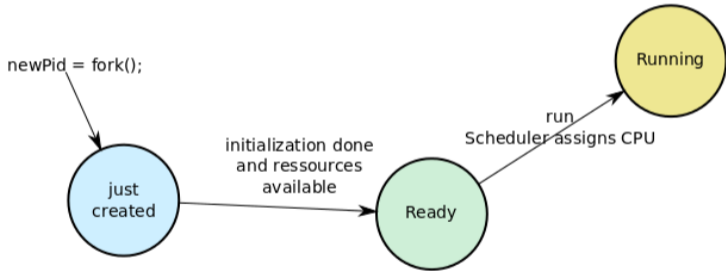


`newPid = fork();`

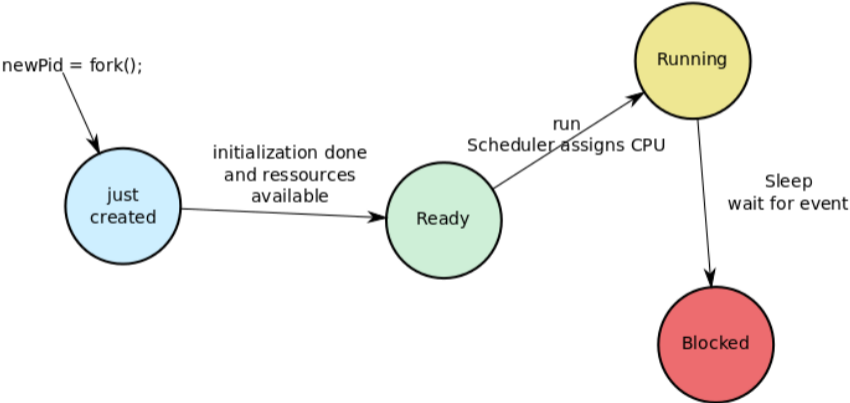




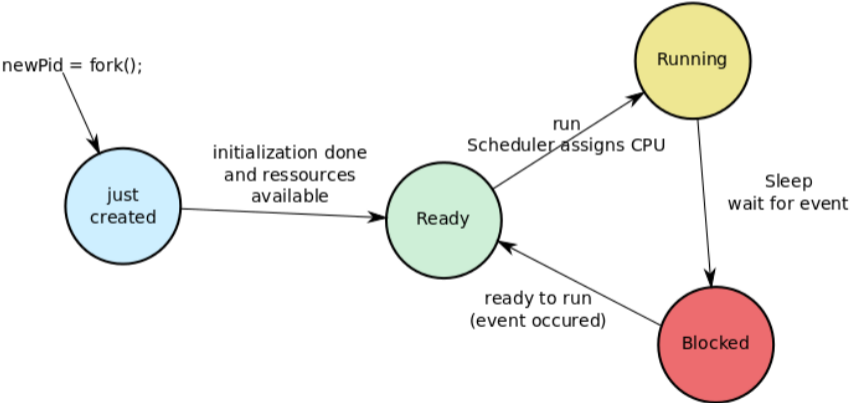
Thread states



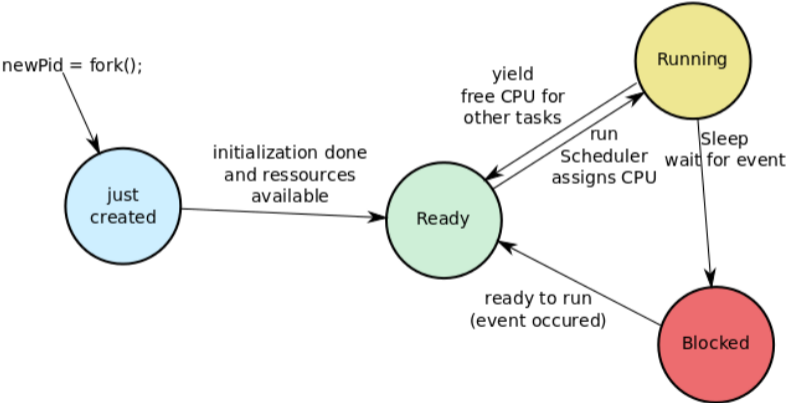
Thread states



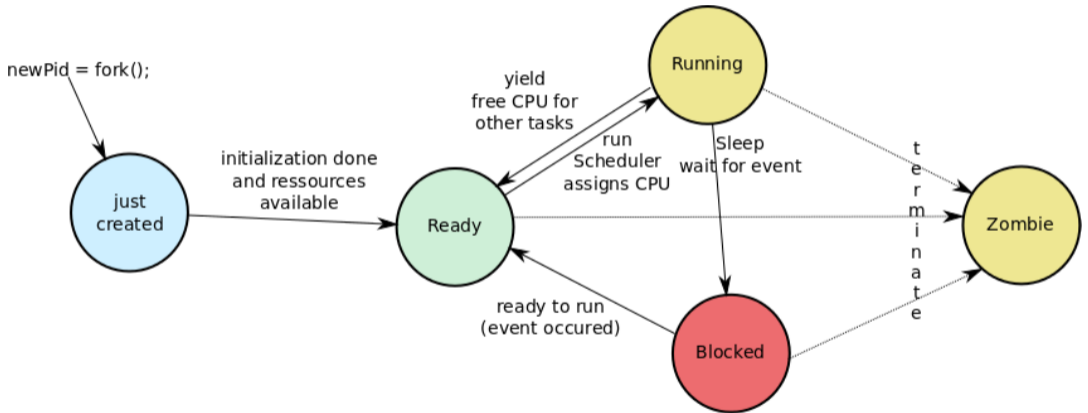
Thread states

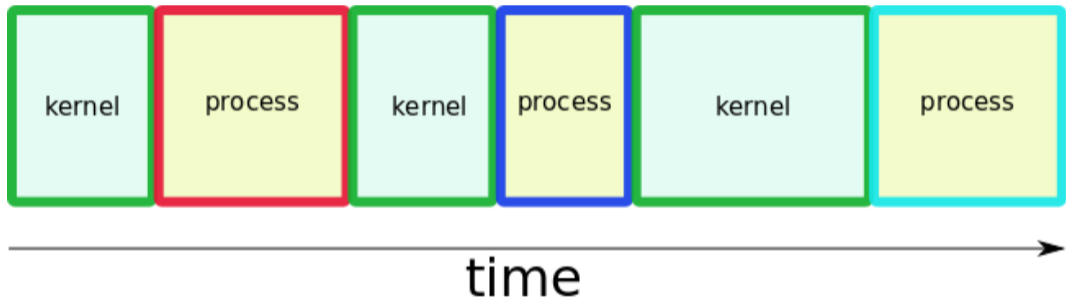


Thread states



Thread states





- at boot time (kernel threads, init processes)
- **at request of a user (how?)**
 - also: start of a scheduled batch job (cronjob, how?)

via Syscall!

- UNIX/Linux: `fork` (exact copy)
- Windows: `CreateProcess` (new image)
- SWEB: `fork` (as soon as you have implemented it)

What does the fork do?



Check <http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html!!>

<http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html>:

```
pid_t fork(void);
```

The fork() function shall create a new process. The new process (child process) shall be an **exact copy** of the calling process (parent process) **except** as detailed below:

- unique PID
- copy of file descriptors
- semaphore state is copied
- shall be created with a single thread. If a multi-threaded process calls fork(), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources.
- parent and the child processes shall be capable of executing independently before either one terminates.

<http://pubs.opengroup.org/onlinepubs/9699919799/functions/fork.html>:

```
pid_t fork(void);
```

Upon successful completion, `fork()` shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the `fork()` function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and `errno` shall be set to indicate the error.

```
pid_t child_pid;
child_pid = fork();
if (child_pid == -1) {
    printf("fork failed\n");
} else if (child_pid == 0) {
    printf("i'm the child\n");
} else {
    printf("i'm the parent\n");
    waitpid(child_pid, 0, 0); //
        wait for child to die
}
```

- child does not know the parent
- parent knows the child
- parent waits for child to die
(waitpid)

- Normal exit (return value: zero)
- Error exit (return value: non-zero)
- Fatal error (e.g. segmentation fault)
- Killed by another process

Some operating systems have hierarchies:

- implicit hierarchy from forking
- process groups in UNIX/Linux
- doesn't exist in Windows

Implicit parent-child hierarchy on Unix/Linux:

- when parent dies, all children, grand-children, grand-grand-children, . . . , die aswell
- UNIX/Linux also cheats a bit: parent process typically inherits a processes' children, etc.

```
git grep TODO | sort
```

- sort has to wait for input
- what does the sort do in the meantime?
 - loop and check (busy wait)
 - sleep and get woken up
- blocking the process makes sense
- do we actually block the process?

- Processes divide resources amongst themselves (except processor time)
- Threads divide processor time amongst themselves (and a few resources)
- Building block of modern multi-threading are context switches
- Operating system creates illusions
 - for the hardware: there is only 1 thread and a lot of interrupts
 - for the userspace: we can have an arbitrary number of threads