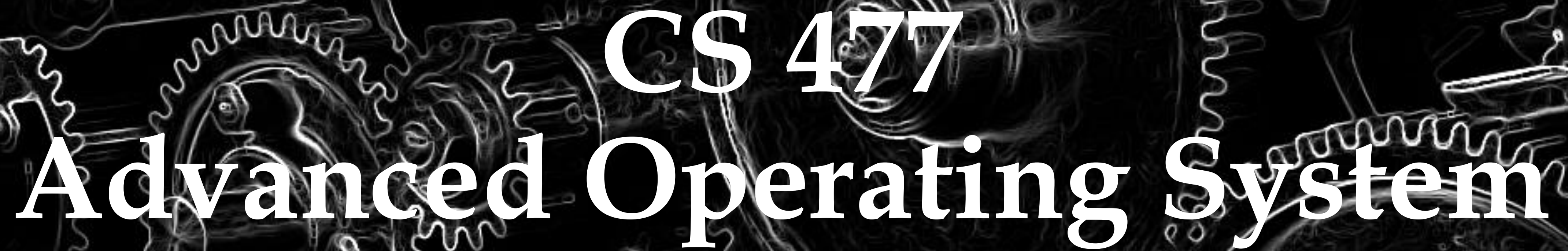


Lecture 05: Process Management

Before getting started …

- Time for the **project** team by the end of this week
  - Received two project ideas (will discuss in tutorial)
  - Deadline for project proposal: Oct 17

EPFL

Until now …

Lectures

- System call: interface between application and kernel
- Kernel data structures & synchronization primitives

Practical:

- Getting, building, and exploring the Linux kernel code
- Kernel modules and debugging

EPFL

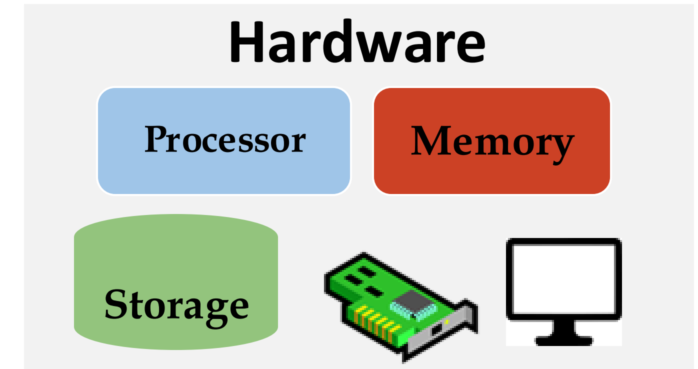
Focus of today's lecture

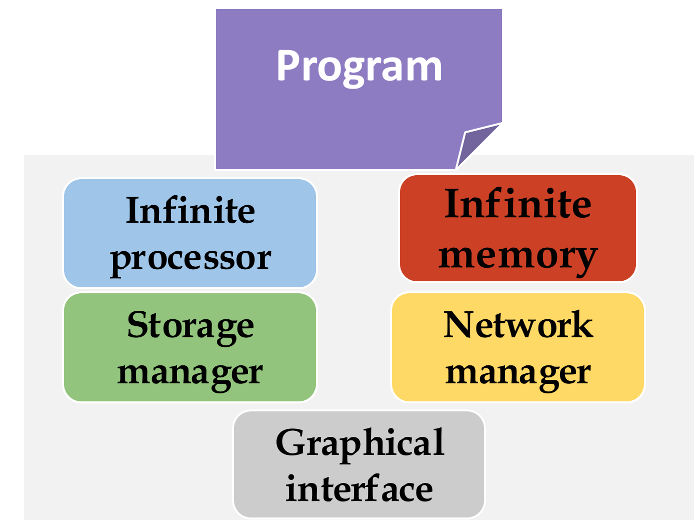
- Process and its management in the Linux kernel
  - The process descriptor: **`task_struct`**
  - Process lifecycle
    - Process creation
    - Process termination
  - Threads

EPFL

What is a process?

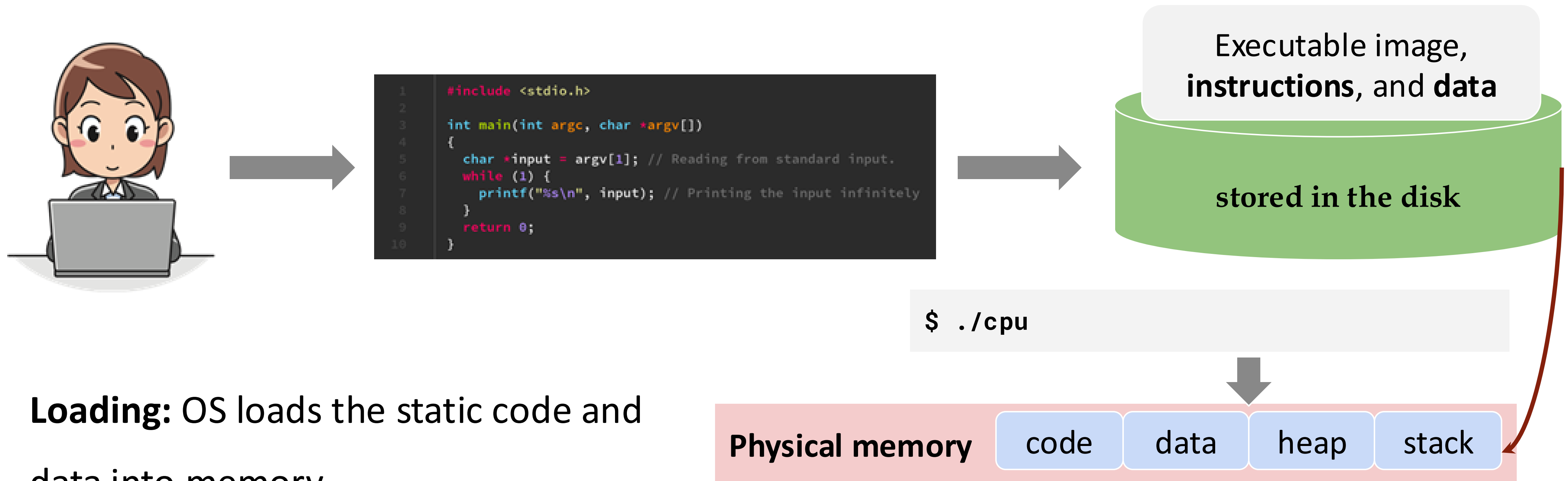
- Process → OS abstraction of a running program
  - Provides *isolation* and *virtualization of hw resources*
- Each process behaves as if it owns:
  - CPU, private memory, own IO devices
- Process components:
  - ***Execution streams*** → sequence of instructions being executed; several streams (threads)
  - ***Process state*** → Everything a stream depends on:
    - *CPU registers; virtual address space; open files, sockets and other kernel resources*





EPFL

How does the OS create a process?



- **Loading:** OS loads the static code and data into memory
- **Memory allocation:** Allocate process memory regions (heap and stack)
- **Initialization:** Initialize tasks related to IO (setting up STDIN, STDOUT, STDERR)
- **Ready:** OS sets the stage for running the process by transferring the CPU control at beginning of the program's entry point (e.g., main() function)

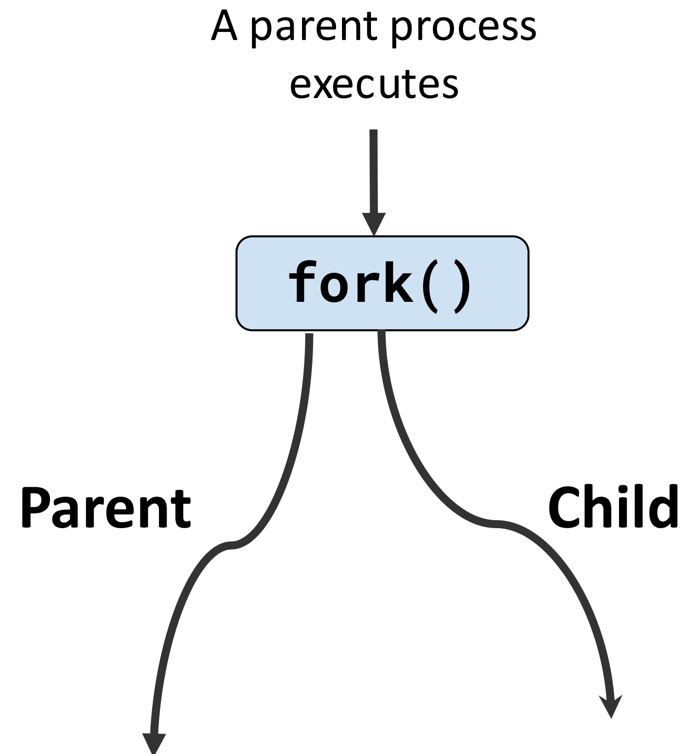
EPFL

Process view from user space

- OS provides a set of interfaces to create and manage processes:
  - `fork()` executes a child process (a copy of the parent process)
  - `exec()` executes a new program
  - `exit()` terminates the current process
  - `wait()` blocks the current process until the child terminates

- This is a small subset of complex process APIs

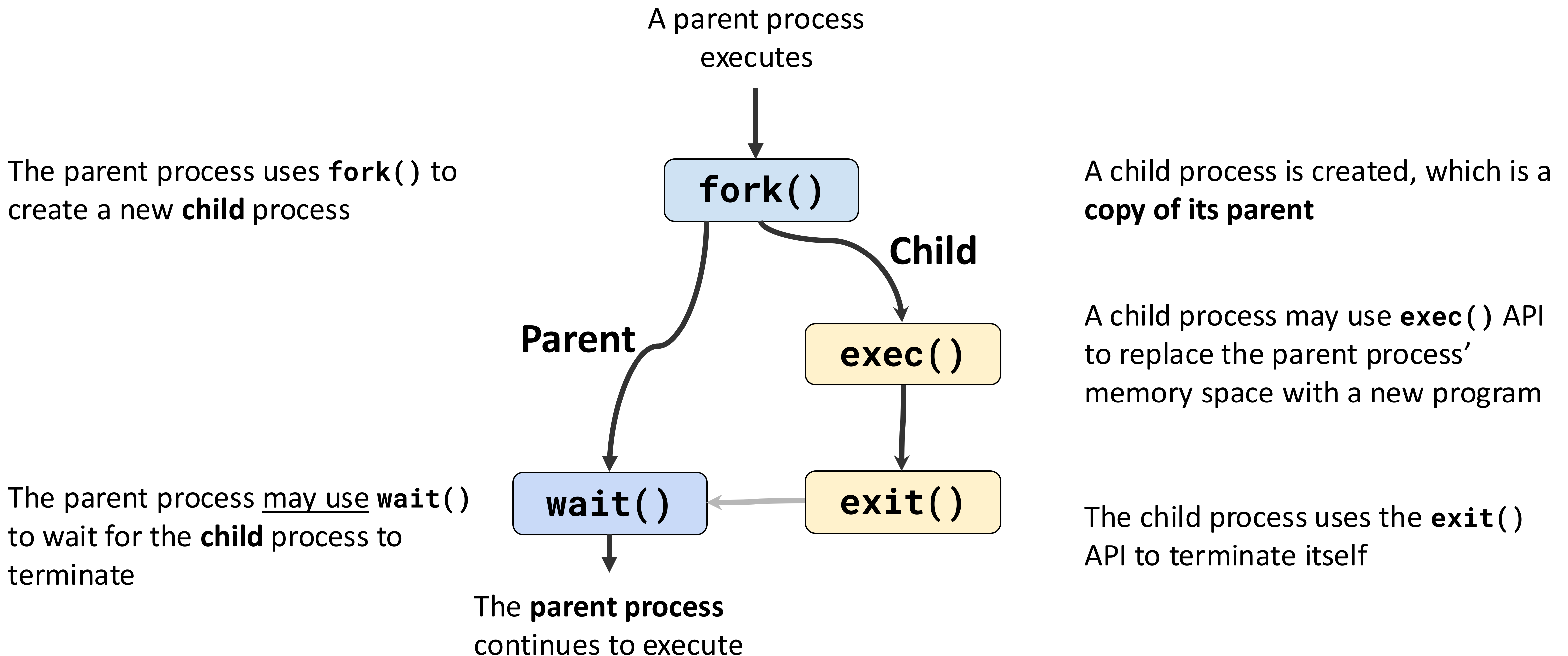
Process view from user space



A new process (the child process) is created. The new **child** is the **copy of** the **parent process**.

EPFL

Process view from user space



Q. Why do we have a separate fork() and exec()?

fork() example

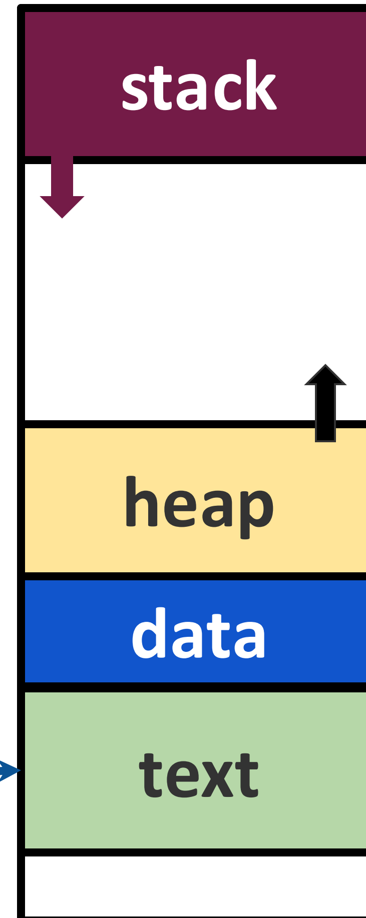
```c
int main(void) {
  pid_t pid;
  int wstatus, ret;

  pid = fork(); /* create a child process */
  switch(pid) {
    case -1: /* fork error */
      perror("fork");
      return EXIT_FAILURE;
    case 0: /* pid = 0: new born child process */
      sleep(1);
      printf("Noooooooo!\n");
      exit(99);
    default: /* pid = pid of child: parent process */
      printf("I am your father!: your pid is %d\n", pid);
      break;
  }
  /* A parent wait until the child terminates */
  ret = waitpid(pid, &wstatus, 0);
  if(ret == -1)
    return EXIT_FAILURE;
  printf("Child exit status: %d\n", WEXITSTATUS(wstatus));
  return 0;
}
```

Q. What is the output?

EPFL

Process representation in Linux: task_struct

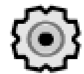
```
/* linux/include/linux/sched.h */
struct task_struct {
  struct thread_info      thread_info;     /* thread information */
  volatile long           __state;         /* task status: TASK_RUNNING, etc */
  void                    *stack;          /* stack of this task */
  int                     prio;            /* task priority */
  struct sched_entity     se;              /* information for processor scheduler */
  cpumask_t               cpus_mask;       /* bitmask of CPUs allowed to execute */
  struct list_head        tasks;           /* a global task list */
  struct mm_struct        *mm;             /* memory mapping of this task */
  struct task_struct      *parent;         /* parent task */
  struct list_head        children;        /* a list of child tasks */
  struct list_head        sibling;         /* siblings of the same parent */
  struct files_struct     *files;          /* open file information */
  struct signal_struct    *signal;         /* signal handlers */

  /* ... */
  /* NOTE: In the Linux kernel, process and task are used interchangeably. */ };
  /* Lets use pahole to see the structure. */
  /* Lets check the whole status of a process maintained: `cat /proc/<pid>/status` */
};
```



PC: Program counter; SP: Stack pointer

Process representation in Linux: `task_struct`

- Each process (or thread) is represented by **`task_struct`**
- Stores all runtime information about a task → scheduling, CPU, memory, file

 **Memory placement**

- Dynamically allocated (on the heap)
  - Not on the kernel stack: avoids corruption from stack overflow or exploits

Accessing the current task

  - Kernel keeps a per-CPU pointer to the currently running **`task_struct`**

```
/* linux/arch/x86/include/asm/current.h */
DECLARE_PER_CPU(struct task_struct *, current_task);
static __always_inline struct task_struct *get_current(void)
{
    return this_cpu_read_stable(current_task);
}
#define current get_current() /* TODO: Let's check how `current` is used. */
```

Process identifier (PID): pid_t

- Maximum size is 32768 (**`int`**)
- Can be increased to 4 millions
- Wraps around when maximum value is reached

EPFL

Process status: `task->_state` (running / sleeping)

- **`TASK_RUNNING`**
  - A task is runnable (running or in a per-CPU scheduler run-queue)
  - A task could be in user- or kernel-space
- **`TASK_INTERRUPTIBLE`** (blocking syscalls, sleeping)
  - Process is suspended (sleeping) until some condition becomes true
  - Switched to **`TASK_RUNNING`** when the waiting condition becomes true or a signal is received

  → Used when a task should have a chance to cancel or interrupt a blocked op
- **`TASK_UNINTERRUPTIBLE`** (wait for IO completion, memory reclaim)
  - Same as **`TASK_UNINTERRUPTIBLE`** but does not wake up on a signal

  → Used when interruption can lead to **corrupt** state or make device/resource inconsistent

EPFL

Process status: `task->_state` (intentionally pasued)

- **`__TASK_TRACED`**
  - Traced by another process (ex. debugger)
  - **`ptrace()`** system call
- **`__TASK_STOPPED`**
  - Task has been stopped (paused) by a signal
    - **`SIGSTOP, SIGTSTP`**

  (when you press Ctrl^Z)

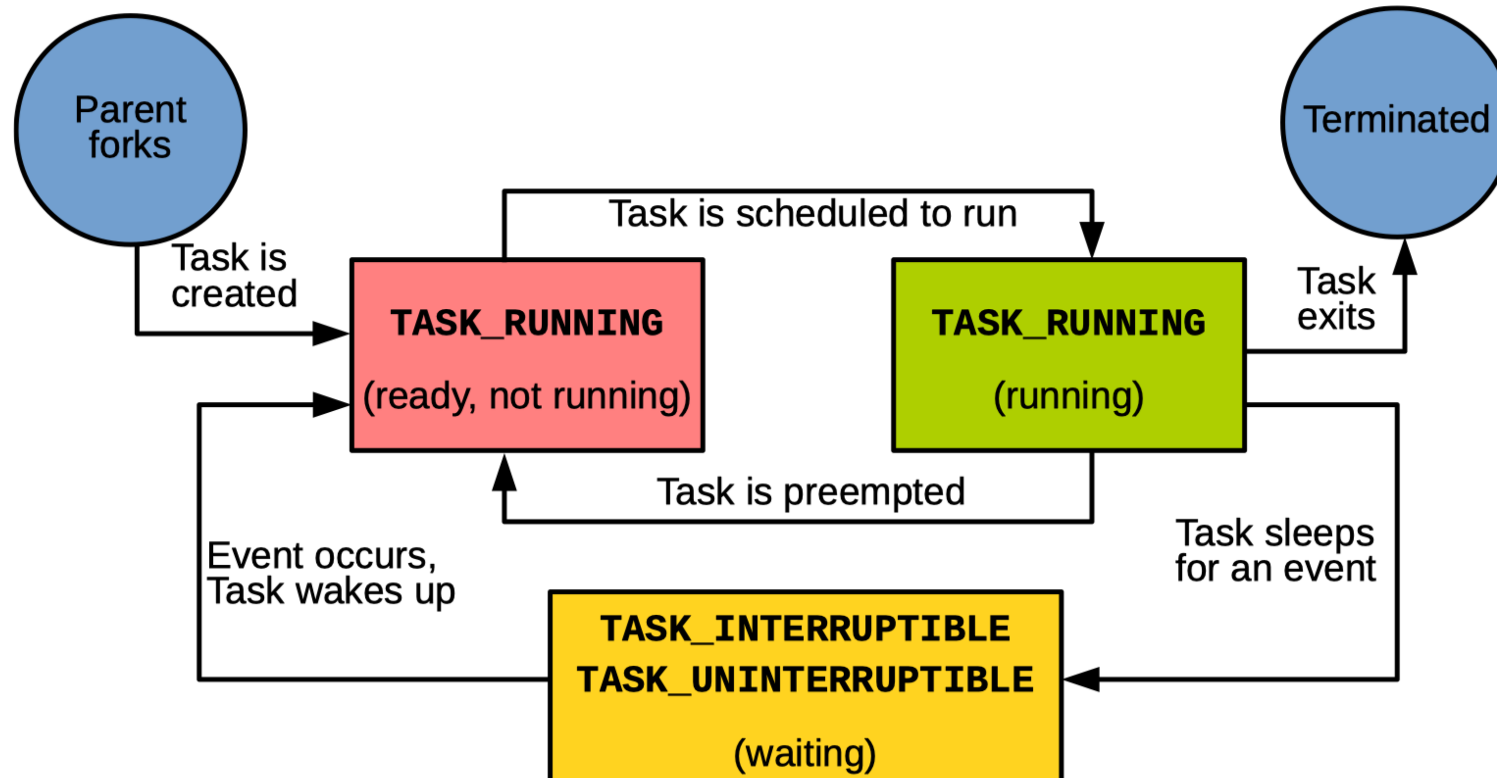
- Both require an explicit continue signal

  (**`SIGCONT`** or **`ptrace(PTRACE_CONT)`**)

Q. What happens if a task receives a signal like STOP/KILL etc?

EPFL

Process status transition



Example: sleep/wakeup using task states

- Producer:
  - Generates an event and wakes up a consumer
- Consumer:
  - Checks if there is an event
  - If so, process all pending events in the list
  - Otherwise, sleeps until the producer wakes the consumer up
- Needs:
  - Handle concurrency between a consumer and a producer
  - Add/del elements to/from a list (events)
  - Handle sleeping of a consumer

EPFL

Example: sleep/wakeup using task states

Producer task:

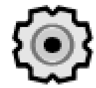
```
001  spin_lock(&list_lock);
002  list_add_tail(&list_head, new_event);  /* append an event to the list */
003  spin_unlock(&list_lock);
004  wake_up_process(consumer_task);        /* and wake up the consumer task */
```

Consumer task:

```
100  set_current_state(TASK_INTERRUPTIBLE); /* set status to TASK_INTERUPTIBLE */
101  spin_lock(&list_lock);
102  if(list_empty(&list_head)) {           /* if there is no item in the list */
103          spin_unlock(&list_lock);
104          schedule();                /* sleep until the producer task wakes this */
105          spin_lock(&list_lock);     /* this task is waken up by the producer */
106  }
107  set_current_state(TASK_RUNNING); /* change status to TASK_RUNNING */
108
109  list_for_each(pos, list_head) {
110          list_del(&pos)
```

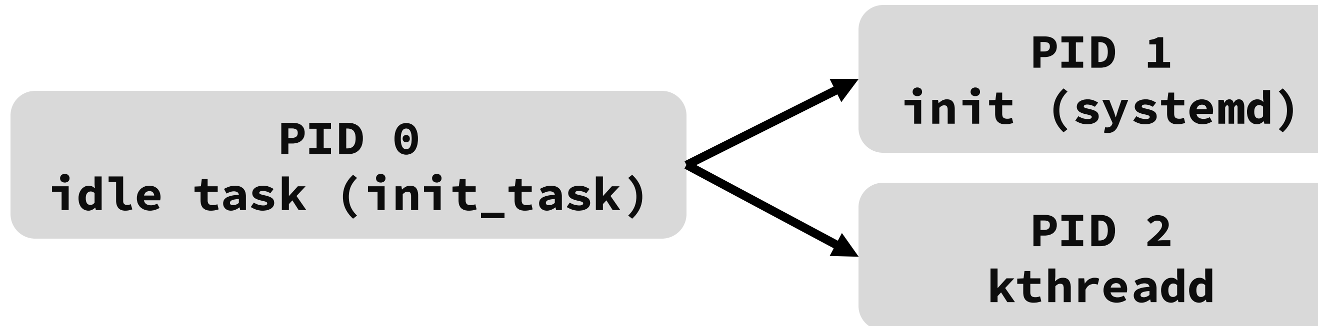
EPFL

Kernel execution context

 Two contexts in which the kernel runs

1. Process context → serving a process (application)
   - Kernel executes **on behalf of a process**
   - **`current`** points to that process' **`task_struct`**
   - Kernel can sleep, schedule or block safely
   - Ex: system call, io control (ioctl)
2. Interrupt context → serving hardware
   - Kernel executes **in response to an interrupt** (hardware)
   - Not associated with any user process → **`current`** may be meaningless
   - **Cannot sleep** or call blocking functions
   - Ex: device handler, network deferred handler (**`softirq`**), timer handler

Process hierarchy in the Linux kernel



Property	PID 0 (swapper)	PID 1 (init)	PID 2 (kthreadd)
Created by	static kernel allocation (`init_task()`)	Forked by `rest_init()`	Forked by `rest_init()`
Context	Kernel-only	Starts as kernel thrad → become user space init	Kernel-only
Run as	CPU idle loop	System init, service launcher	Kernel thread manager
Function	Keeps CPU busy / power-save when idle	Root of user-space	Parent of all kernel threads

Let's check the process tree (`pstree`)

Process hierarchy in the Linux kernel

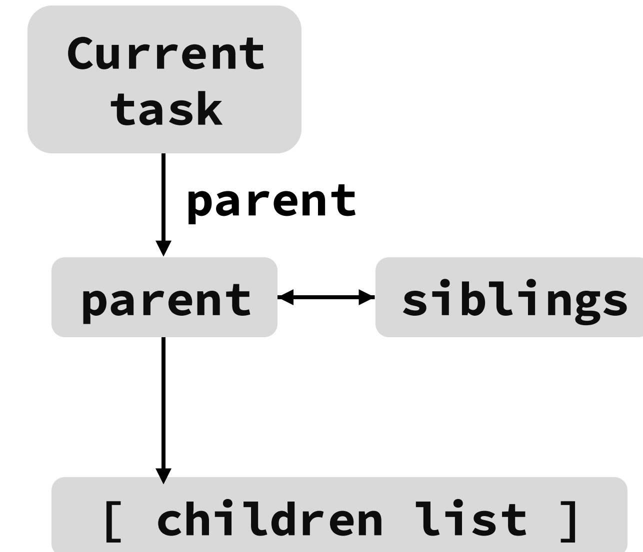
```
systemd-+-ModemManager---3*[{ModemManager}]
        |-accounts-daemon---3*[{accounts-daemon}]
        |-at-spi-bus-laun-+-dbus-daemon
        |                 `-4*[{at-spi-bus-laun}]
        |-at-spi2-registr---3*[{at-spi2-registr}]
        |-automount---3*[{automount}]
        |-avahi-daemon---avahi-daemon
        |-blkmapd
        |-colord---3*[{colord}]
        |-cron
        |-cups-browsed---3*[{cups-browsed}]
        |-cupsd---dbus
        |-dbus-daemon
        |-fail2ban-server---4*[{fail2ban-server}]
        |-fwupd---3*[{fwupd}]
        |-gdm3-+-gdm-session-wor-+-gdm-x-session-+-Xorg---3*[{Xorg}]
        |      |                 |               |-dbus-run-sessio-+-dbus-daemon
        |      |                 |               |                 `-gnome-session-b-+-gnome-shell-+-ibus-daemon-+-ibus-dconf---4*[+
        |      |                 |               |                                   |             |             |-ibus-engine-sim-+
        |      |                 |               |                                   |             |             `-3*[{ibus-daemon}+
        |      |                 |               |                                   |             |-mutter-x11-fram---6*[{mutter-x+
        |      |                 |               |                                   |             `-13*[{gnome-shell}]
        |      |                 |               |                                   |-gsd-a11y-settin---4*[{gsd-a11y-settin}]
        |      |                 |               |                                   |-gsd-color---4*[{gsd-color}]
        |      |                 |               |                                   |-gsd-datetime---4*[{gsd-datetime}]
        |      |                 |               |                                   |-gsd-housekeepin---4*[{gsd-housekeepin}]
        |      |                 |               |                                   |-gsd-keyboard---4*[{gsd-keyboard}]
        |      |                 |               |                                   |-gsd-media-keys---4*[{gsd-media-keys}]
        |      |                 |               |                                   |-gsd-power---4*[{gsd-power}]
        |      |                 |               |                                   |-gsd-print-notif---3*[{gsd-print-notif}]
        |      |                 |               |                                   |-gsd-rfkill---3*[{gsd-rfkill}]
        |      |                 |               |                                   |-gsd-screensaver---3*[{gsd-screensaver}]
        |      |                 |               |                                   |-gsd-sharing---4*[{gsd-sharing}]
        |      |                 |               |                                   |-gsd-smartcard---4*[{gsd-smartcard}]
        |      |                 |               |                                   |-gsd-sound---4*[{gsd-sound}]
        |      |                 |               |                                   |-gsd-wacom---3*[{gsd-wacom}]
        |      |                 |               |                                   `-4*[{gnome-session-b}]
        |      |                 |               `-3*[{gdm-x-session}]
        |      |                 `-3*[{gdm-session-wor}]
        |      `-3*[{gdm3}]
```

EPFL

Process relationships in the kernel

- **fork**-based process creation

Relationship	Pointer / field	Meaning
Parent	**current→parent**	Task that created me
Children	**current→children**	List of my child tasks
Siblings	**current→sibling**	Other children of my parent
Global list	**current→tasks**	Linked list of all tasks in the system



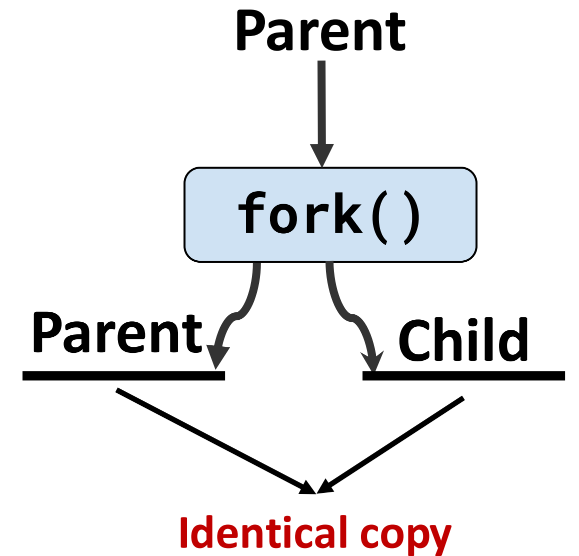
- Traversal helpers:

Macro	Purpose
next_task(t)	Iterate over the list of tasks
for_each_process(t)	Safely traverse every process in the system

include/linux/sched/signal.h

Process creation overview

- Linux does not create a task from scratch (unlike **spawn** on Windows)
- Relies on **fork()** and **exec()**
  - **fork()** creates a child, a copy of the parent process
    - Only PID, PPID, and some resources differ (duplicates process)
  - **exec()** replaces memory image with a new program



Q. How to implement fork()?

`fork()` implementation

- The expensive approach?

1. Create a new **`task_struct`** for child
2. Copy all memory pages from the parent → child
3. Copy file descriptor table
4. Copy signal handlers
5. Set up page tables pointing to the new pages

Q. What is the issue with this implementation?

EPFL

Naive `fork()` issues

- Extremely slow for large processes
- Wastes memory (copies never used)
- Common pattern: **`fork()`** → **`exec()`**
- Child replaces memory anyway!
- Example:
  - Parent memory use: 1 GB
  - Naive **`fork()`**: Copy all 1 GB
  - Child calls **`exec()`**
  - All 1 GB data is discarded

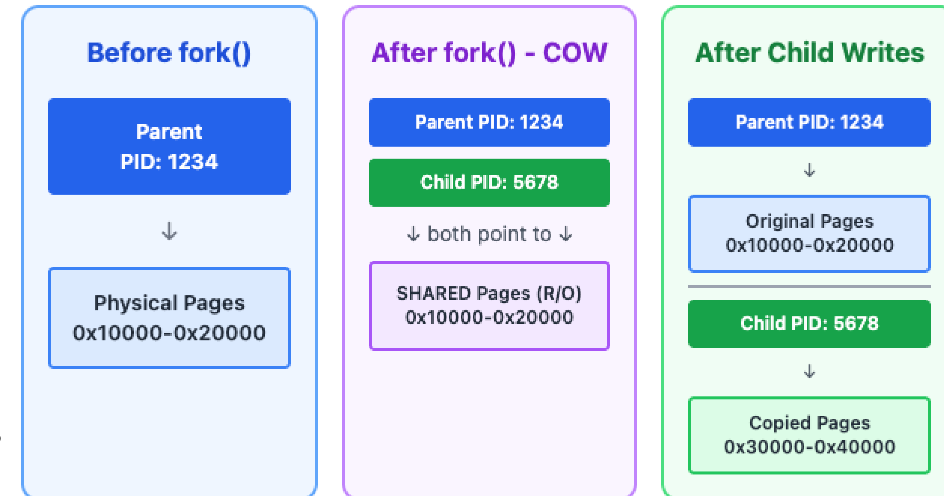
Q. What would be the approach to fix this excessive memory copy issue?

EPFL

`fork()` using Copy-on-write (COW)

Don't copy memory pages <u>immediately</u>; logically duplicate them

- `fork()` logically duplicates the parent without copying pages
- All pages are marked **read-only and shared**
  - Change page table access bits to *read-only*
- When either process writes, create a **private copy**
  - Copy the page and change **page table entry** to *read-write* **(PTE)**



- `fork()` is **fast** by delaying or preventing data copy
- `fork()` **saves memory** by sharing read-only pages among children

EPFL

Advantages of COW

Without COW	With COW
1 GB process fork: ~500 ms	1 GB process fork: ~50µs
Memory copied: 1 GB	Memory copied: 16 KB
Pages allocated: 262,144	Pages allocated: 4 (page table)

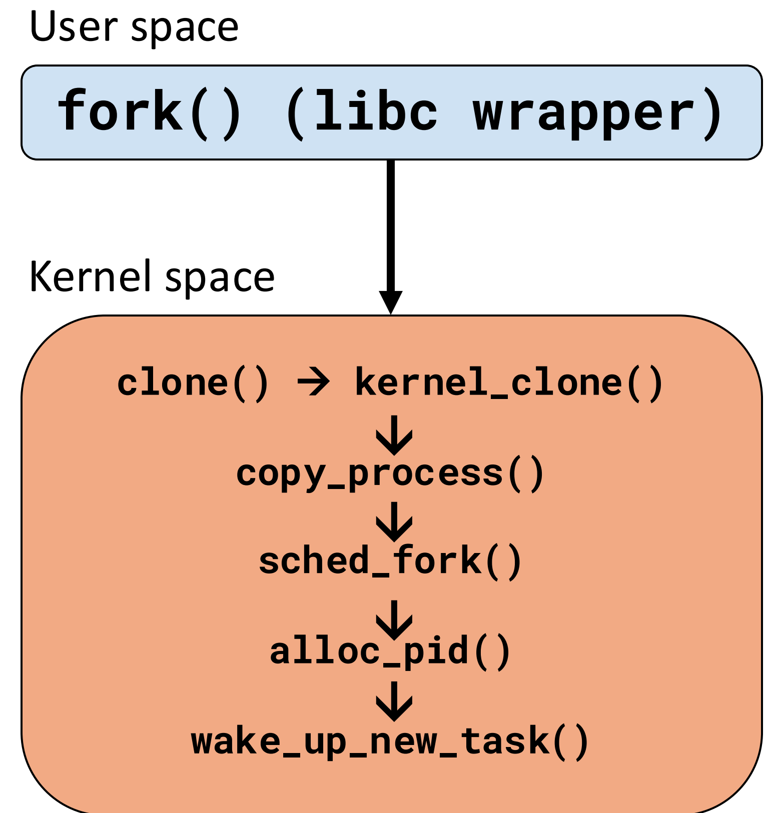
- Real-world impact:
  - Shells fork+exec for every command would be unbearably slow without COW
  - Page serving (Apache nginx) fork worker processes constantly
  - Database servers (PostgreSQL) fork for each connection

COW makes these patterns practical and efficient

EPFL

How **fork()** works internally?

- **fork()** → implemented via **clone()** syscall
- **kernel_clone()** sets up flags, stack, and parent / child linkages
- **copy_process()** duplicates **task_struct**, kernel stack, and signal handlers
- **sched_fork()** assigns to a CPU scheduler
- **alloc_pid()** assigns a new PID
- **wake_up_new_task()** makes the child runnable



EPFL

Process termination overview

- A process terminates via **`exit()`**
  - Often implicit: return from main() → **`sys_exit()`**
- **`sys_exit()`** calls **`do_exit()`**
- **`do_exit()`** does 3 core operations:
  - Release resources (memory, files)
  - Notify parent
  - Transition process to **`EXIT_ZOMBIE`**
- **`do_exit()`** never returns → it releases resources and yields the control back to the scheduler

User space

`exit() (libc wrapper)`

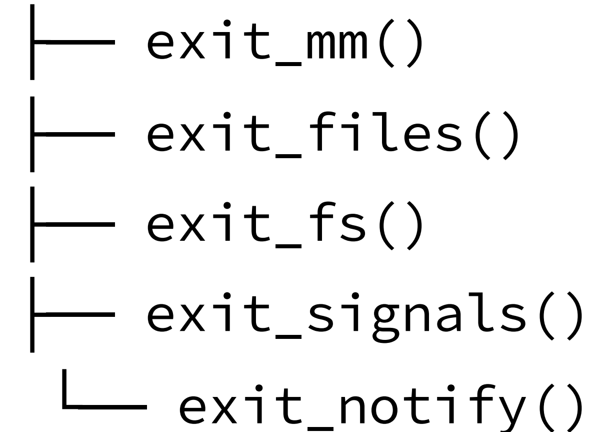
Kernel space

`sys_exit()`
↓
`do_exit()`
↓
`[release resources]`
↓
`[notify parent]`
↓
`state = EXIT_ZOMBIE`

EPFL

Resource cleanup in **do_exit()**

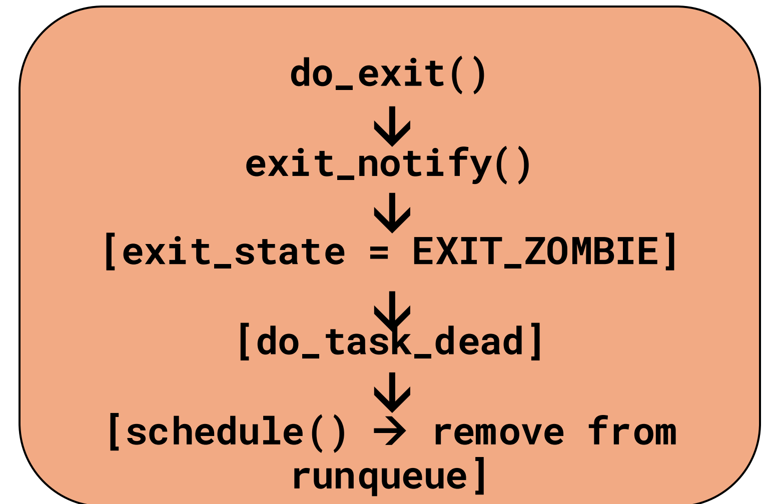
- **exit_mm()** → Release the address space
- **exit_sem()** → Dequeue semaphores if waiting
- **exit_files()** → Drop file descriptor table
- **exit_fs()** → Drop file system info (cwd, root, umask)
- **exit_signals()** → Mark PF_EXITING, clear pending signals



Transitioning to zombie and reaping

- **do_exit()** calls **exit_notify()**
  - Sends **SIGCHLD** to parent
  - Reparents children if parent died (→ **init** or thread group leader)
  - Sets **exit_state = EXIT_ZOMBIE** in **task_struct**
- Then calls **do_task_dead()**
  - Sets **state = TASK_DEAD**
  - Invokes **schedule()** → the task is never run again

Kernel space

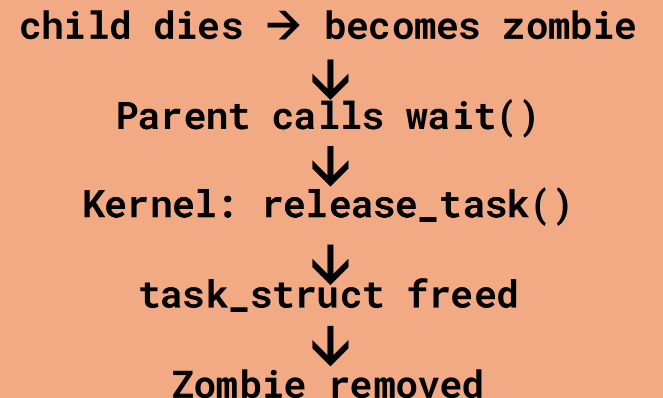


EPFL

Zombie → reaped → fully freed

- Parent collects child's exit status:
  - **waitpid()/wait()** calls **release_task()**
  - The **task_struct** is finally freed
- If the parent dies first:
  - **exit_notify() → forget_original_parent() → find_new_reaper()**
  - The child is reparented to **init** (PID 1)
- A zombie = process has finished execution but not yet reaped
- Reaping ensures proper cleanup of **task_struct** and PID reuse

Kernel space



EPFL

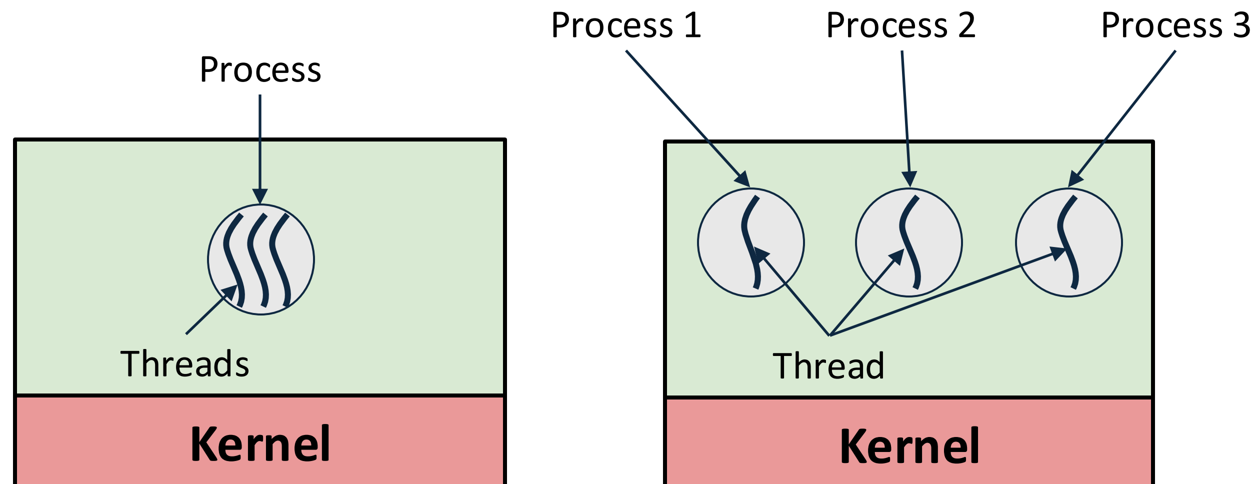
Fork: An abstraction for protection

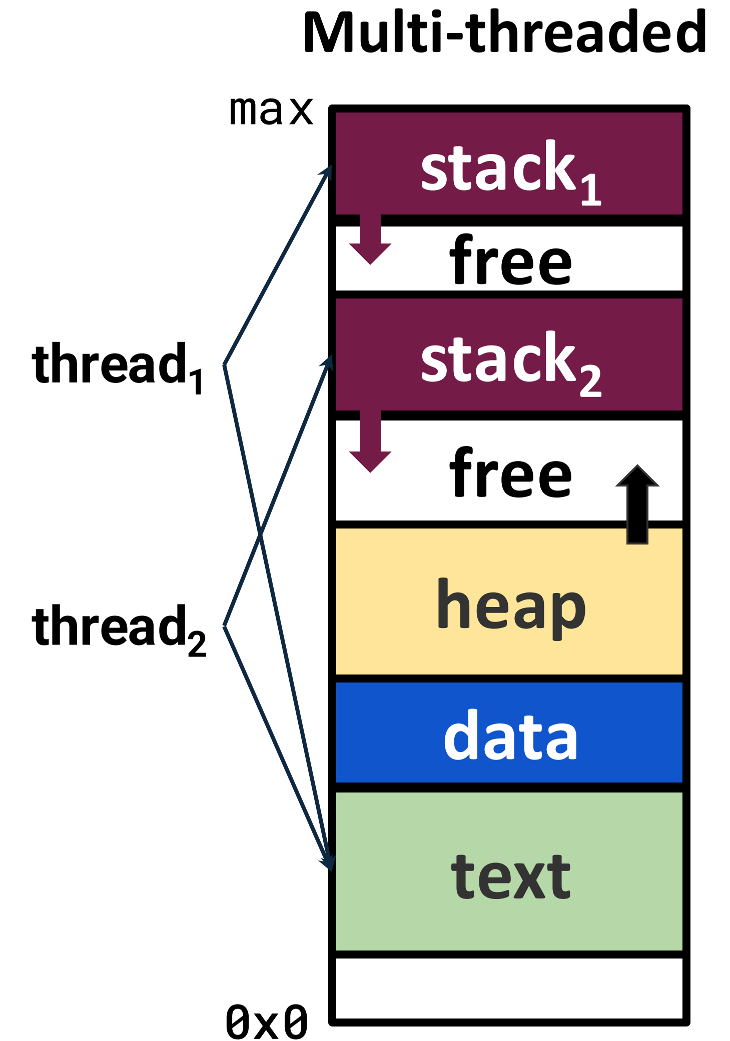
Thread: An execution context

EPFL

Threads

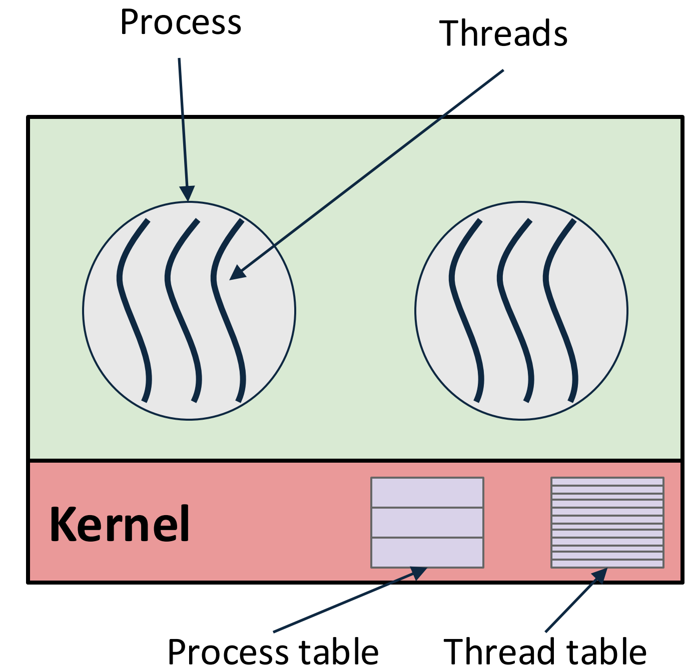
- Thread: execution context + TID + stack
  - Threads share the address space
  - Text, data, heap
- Used for improving concurrency, parallelism, and responsiveness





Kernel-level threads: OS managed

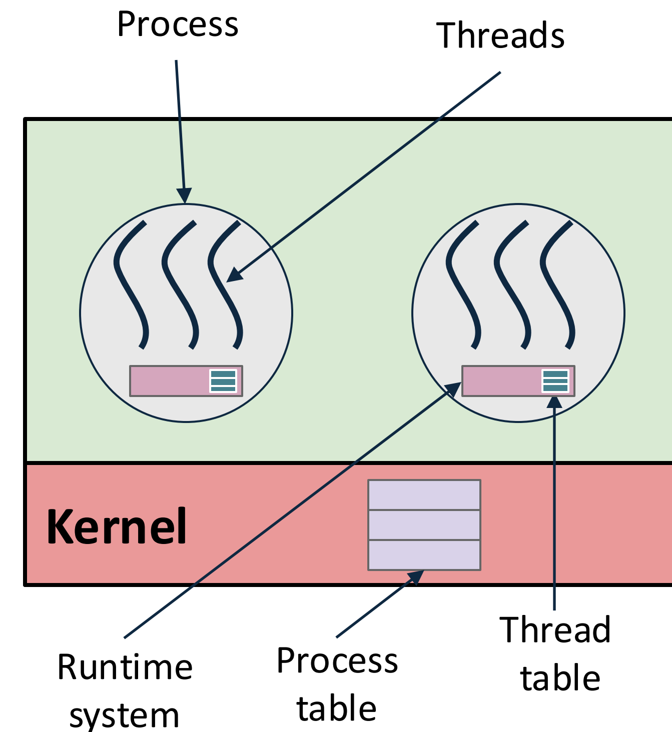
- OS manages threads and processes
- All thread operations are implemented in the kernel
- Thread creation/management requires system calls
- OS schedules all threads
- Creating threads is cheaper than creating processes
- Windows, Linux, Solaris, Mac OS, AIX, HP-AUX



EPFL

User-level threads: runtime/application managed

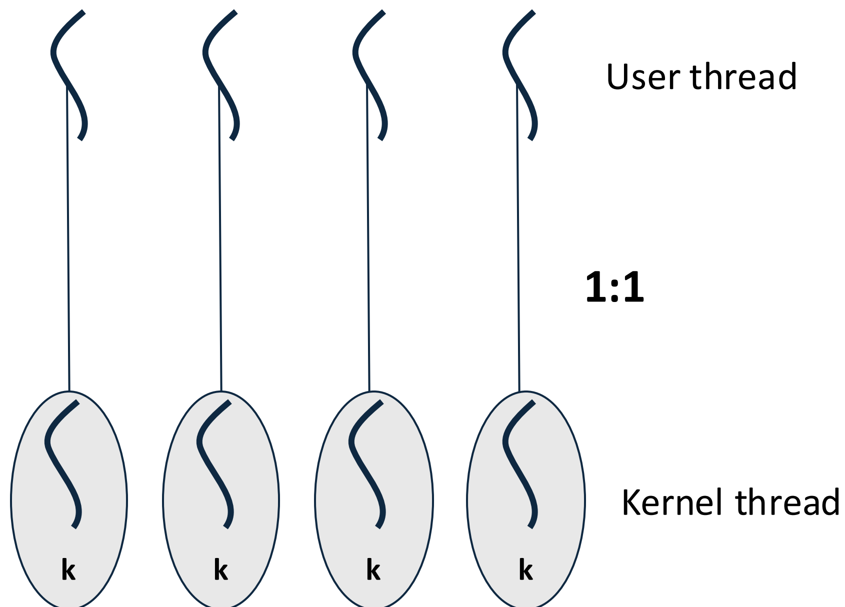
- A library linked into the program, which manages threads
- Threads are invisible to the OS
- All the thread operations are done via procedure calls
  - (no kernel involvement)
- Small and fast:
  - 10–100x faster than kernel-level threads
- Portable
- Tunable to meet application needs
- Java, go, erlang, Node.js, fibers in C++

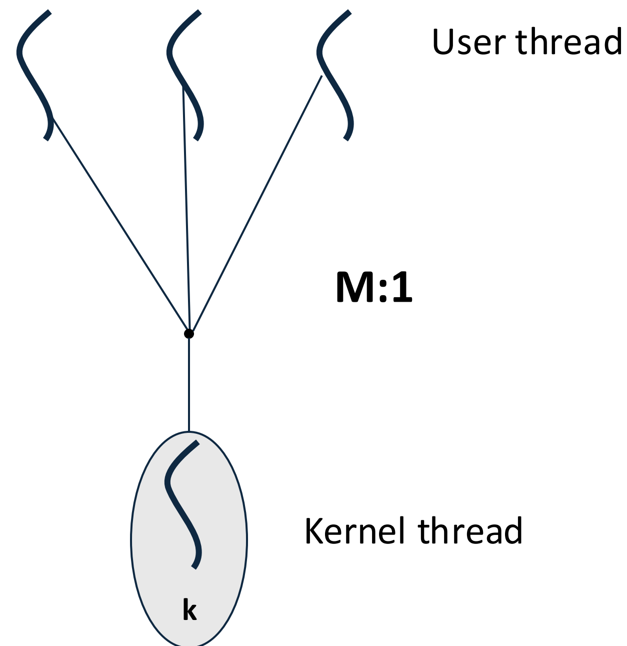


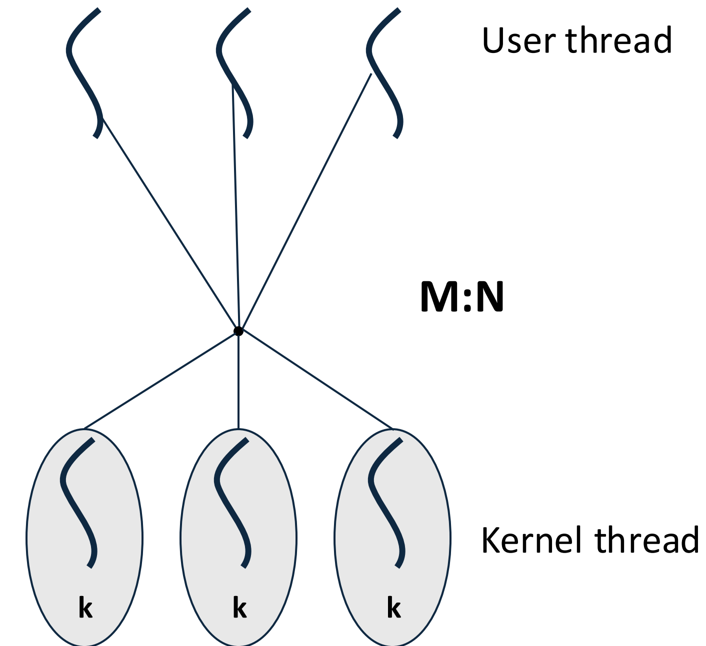
EPFL

Threading models

Model	Mapping	Used in	Status
1:1	One user thread ←→ one kernel thread	Linux, windows	Modern default
M:1	Many user threads ←→ one kernel thread	Solaris green threads, GNU portable threads	Legacy
M:N	many user threads ←→ many kernel threads	Goroutines, Erlang thread pool	Used by languages



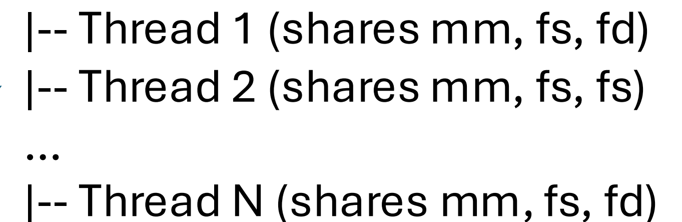




Threads in Linux

- No special "thread" object → only tasks
  - Tasks share parts of their state
- Each thread has its own `task_struct` but may share:
  - Memory (`CLONE_VM`)
  - File system info (`CLONE_FS`)
  - File descriptors (`CLONE_FILES`)
  - Signal handlers (`CLONE_SIGHAND`)

```
clone(CLONE_VM | CLONE_FS |
CLONE_FILES | CLONE_SIGHAND, 0)
```



EPFL

Kernel threads

- Used for background jobs in the kernel
  - Workqueues (kworker)
  - Load balancing (migration)
  - Flushing dirty pages, housekeeping
- Schedulable like regular processes (each has its own `task_struct`)
- Do not have **user address space** (`task_struct→mm = NULL`)
  - Enables fast context switch
  - Not exposed to user programs → safe
  - If mm present → extra memory overhead; risk exposing memory (security)
- Created from **`kthreadd`** kernel thread (PID 2) (**`ps --ppid 2`**)

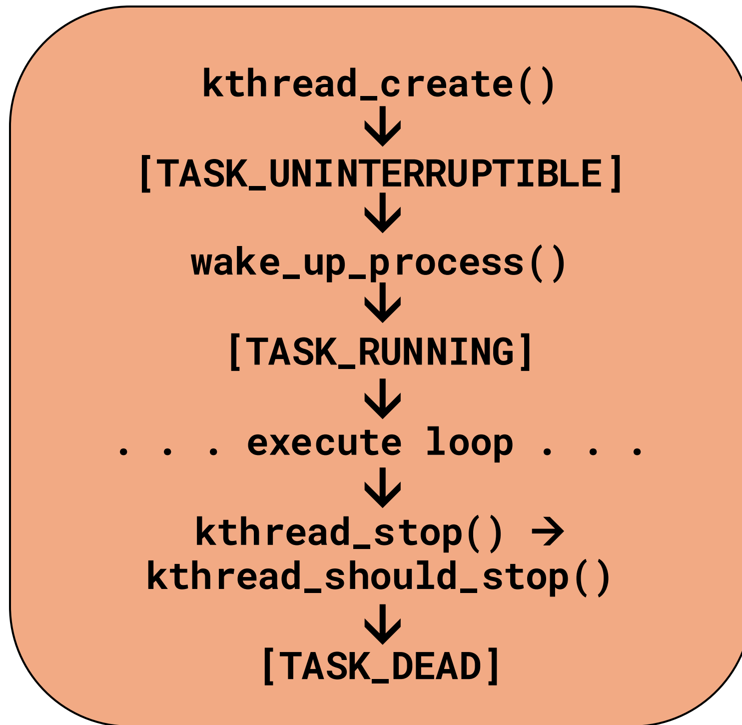
Creating a kernel thread

Function	Purpose
kthread_create(fn, data, name)	Create a kernel thread (not runnable)
wake_up_process()/kthread_run()	Start execution
kthread_stop()	Request a thread to terminate
kthread_should_stop()	Check stop conditions inisde loop

```
struct task_struct *t;

t = kthread_create(worker_fn, NULL, "my_kthread");
wake_up_process(t);

int worker_fn(void *data)
{
    while (!kthread_should_stop())
        do_work();
    return 0;
}
```

EPFL

Kernel thread lifecycle and use



- Persistent background tasks managed entirely in kernel space
- They behave like processes but never enter user space

```c
/* linux/fs/ext4/super.c */
static int ext4_run_lazyinit_thread(void)
{
    ext4_lazyinit_task = kthread_run(ext4_lazyinit_thread,
                             ext4_li_info, "ext4lazyinit");
    /* ... */
}

static int ext4_lazyinit_thread(void *arg)
{
    while (true) {
        if (kthread_should_stop()) {
            goto exit_thread;
        }
        /* ... */

static void ext4_destroy_lazyinit_thread(void)
{
    /* ... */
    kthread_stop(ext4_lazyinit_task);
}

static void __exit ext4_exit_fs(void)
{
    ext4_destroy_lazyinit_thread();
    /* ... */
}

module_exit(ext4_exit_fs)
```

Ext4 file system use kernel threads for file system initialization

EPFL

Further readings

- Kernel Korner - Sleeping in the Kernel
- Exploiting Stack Overflows in the Linux Kernel