Operating Systems Internals – Task Management

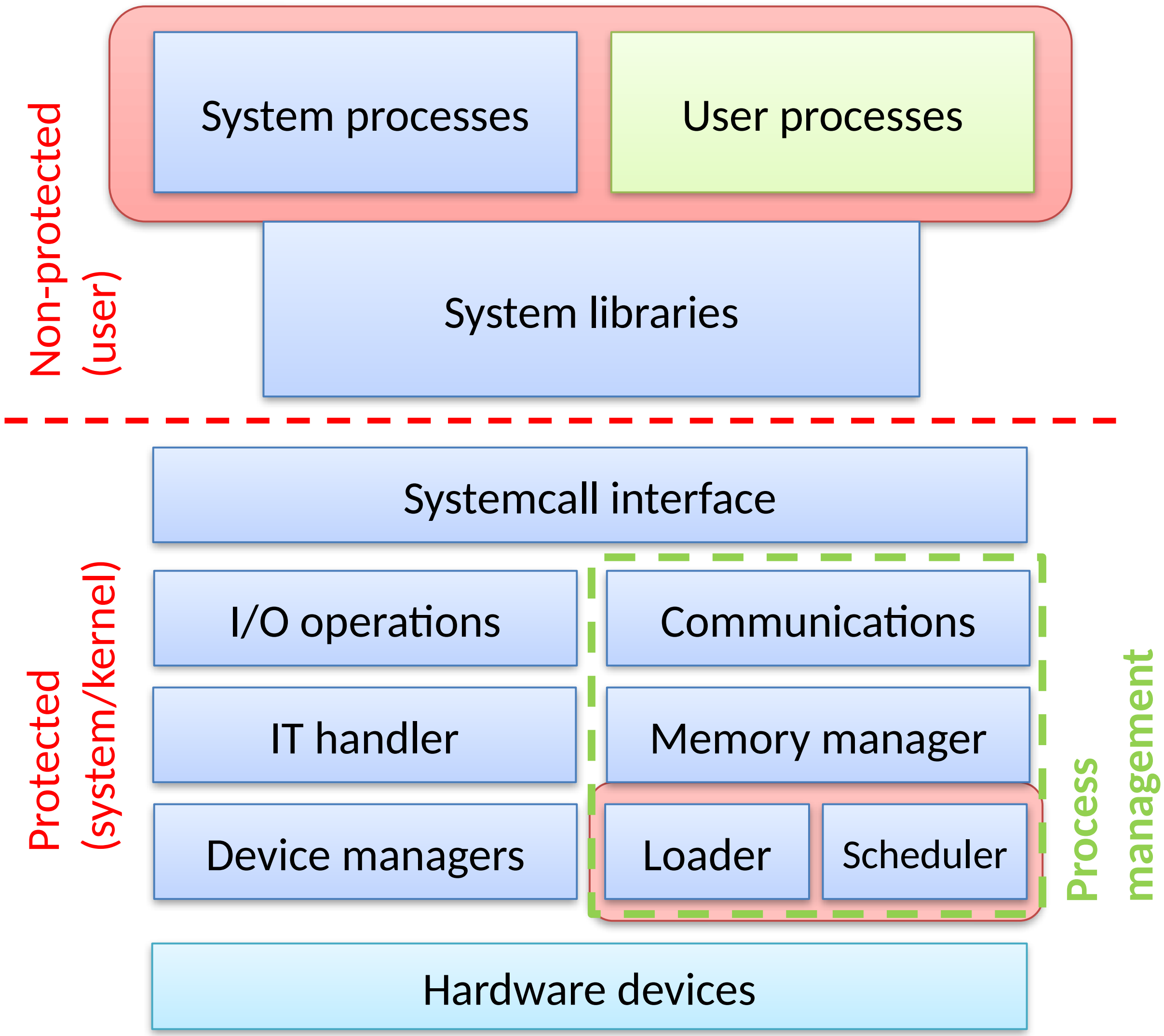
Lecturer: András Millinghoffer
milli@mit.bme.hu

Budapest University of Technology and Economics (BME)
Department of Artificial Intelligence and Systems Engineering (MIT)

The operating systems (recap)

- Serving user (and system) tasks
  - Life-cycle (creation, operation, termination) and event monitoring
  - Providing computational and storage resources
  - Providing access to the devices of the computer
- System libraries: Common functions for applications
  - Supports the application development
  - Providing simple interfaces to system calls (entering protected mode)
- System applications (and services)
  - Applications (user-mode) which come with the OS
  - Integrated commands, user interfaces, services

The main blocks of the OS and the kernel



The nature of user tasks

- Tasks with intensive I/O usage
  - Moving and processing data
  - Reading and writing to HW devices (disc, USB drive, etc.)
  - Most of the time these tasks' state is „waiting/idle"
    - Waiting for I/O operations or user interactions
    - Therefore less CPU time is needed
- Tasks with intensive CPU usage
  - Performing longer computational operations
  - Most of time these tasks' state is „running" (at least want to be...)
  - Compared to CPU usage less I/O is needed
  - E.g.: cryptography, mathematical operations
- Tasks with intensive memory usage
  - Working with large amount of data at once
  - If there is enough memory -> CPU intensive, if not -> I/O intensive
  - E.g.: multiplying large matrices, building and using database indexes
- Special demands (examples)
  - Providing real-time operation
  - Smooth media playback

User expectations about user tasks

- Low waiting times
  - Waiting time
    - Waiting for resources (taken by other tasks), idle state
  - Turnaround time
    - Time that a task needs to finish it's operation
  - Response time
    - Response time to a given event

- Good resource utilization
  - CPU utilization
    - Time ratio of the time, when the CPU is not idle
  - Throughput
    - Tasks performed in given time slice
  - Overhead
    - „Wasting" resources to OS administrative tasks

- Predictability, deterministic operation
  - Small variance of the measures above

The optimal task executor system

- The naive user expects optimal behavior for the OS
  - Executes the users' tasks
  - Minimizing the waiting and response times
  - With good resource (CPU, I/O) utilization
  - With little overhead
- What's he experience using the system?
  - Some tasks run very slow (starving)
  - The concurrent tasks interfere with each other (trying to use the same resources)
  - Some of the applications freeze without any reason
  - Occasionally the whole system becomes unusable (for some time or permanently)
- What's causing these difficulties?
  - The OS does not know the nature of the tasks in advance
  - High number of tasks with different natures
  - The tasks may have explicit or implicit effects on each other
  - The tasks' programs are not optimal, especially in cooperation
  - Occasionally the system is overloaded, the overhead gets high suddenly (thrashing)

The basics of task managing

- The user activities are performed by programs
  - They start, run and terminate
- The **task** is a program during execution
  - The execution is managed by the OS
  - A program stored on the HDD is a static binary program and data structures
  - A task is a dynamic entity with state and life-cycle
  - **State**: The administrative properties of the task in a given moment
  - **Life-cycle**: The state transitions of the task from the start to the termination
- Assigning user activities with tasks
  - In most cases one activity is performed by one task
    - Except some cases: complex activities require more than one task
    - Or parallel tasks (on multiple machines)
  - The task can communicate and cooperate
    - Sending and receiving data from each other
    - The main activity can be decomposed to smaller jobs, partial results can be summarized
    - The tasks can form common procedure structures and cooperation schemas

Separation of the tasks (abstract virtual machine)

- The ideal scenario: every task runs independent of each other
  - No effects on other tasks
  - It seems they running on a separate machine (resources)

- In the reality: not enough resources for each task
  - They have to share the resources (CPU, memory, etc.)
  - Goal: the task (and the user) don’t notice this
  - The kernel provides an **abstract virtual machine** for the tasks (virtual CPU and memory)
  - A typical multi-programmed system
    - M processor (1<= M <= 8), N task (N > 10-100)
    - More task than processor (N >> M)
    - N abstract virtual machines have to be assigned to the physical resources
    - In a way that the tasks don’t the existence of other tasks, but still sharing the common resources

- Complex activities require more than one task: this makes the situation more complex
  - Communication (IPC) and cooperation schemas have to be provided

The base types of tasks: process and thread

- Not every task needs a „full” abstract virtual machine assigned
  - Running of parallel jobs don’t has to be complicated with task-separation
  - The task-separation need higher administrative procedures (higher overhead)
- Process
  - A task with its own memory range, it can contain threads
- Thread
  - A task with sequential operation, it may share memory with other threads
- Relationship between process and threads
  - The process contains threads, which running „parallel”
  - The threads in a process have shared memory (but own stack)
  - They can communicate with each other via the shared memory (variables)
  - There isn’t any memory protection between them, the developer/programmer has to deal with this
  - The threads memory are separated from other process threads’ memory by the OS
  - Communication between processes therefore more complicated

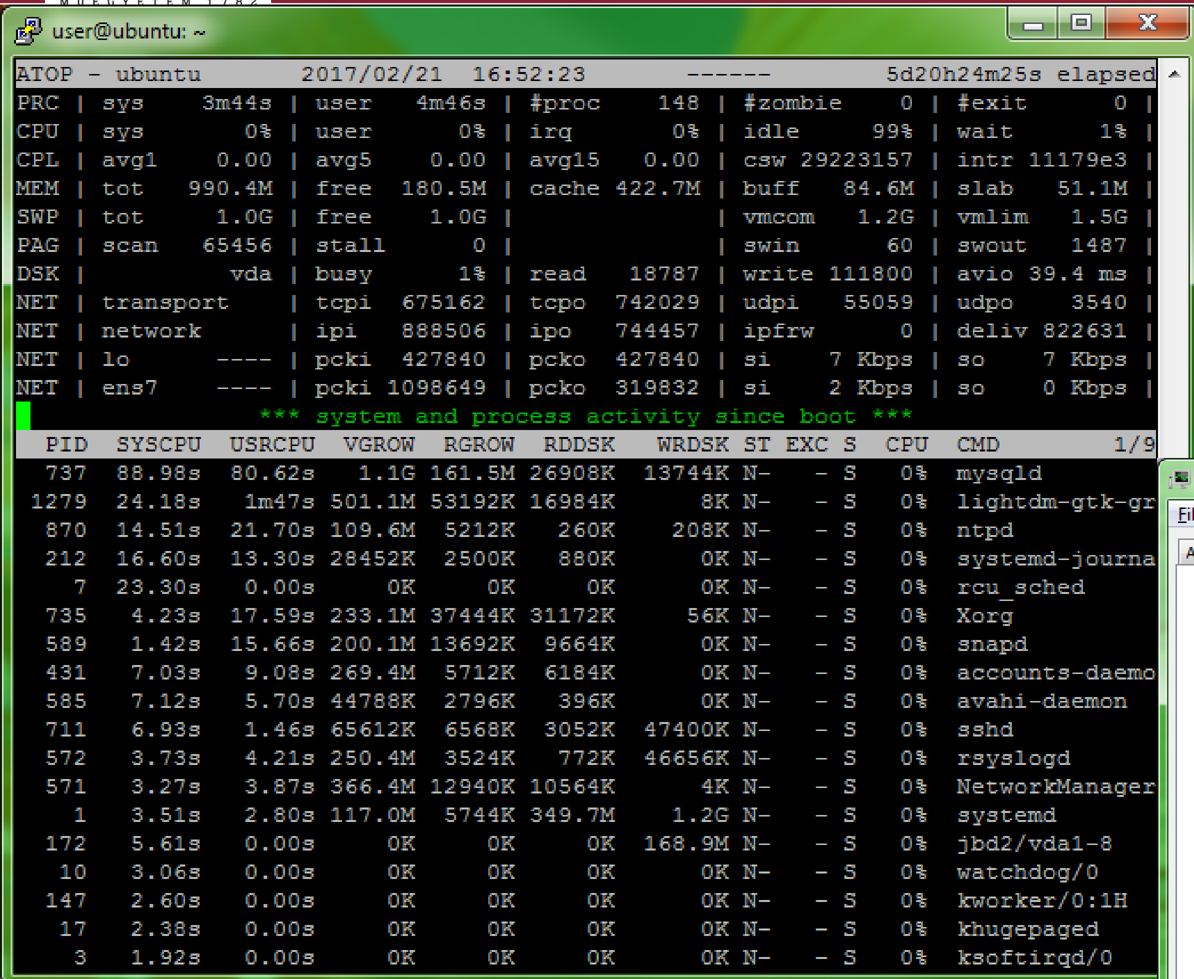
Should I use a process or a thread?

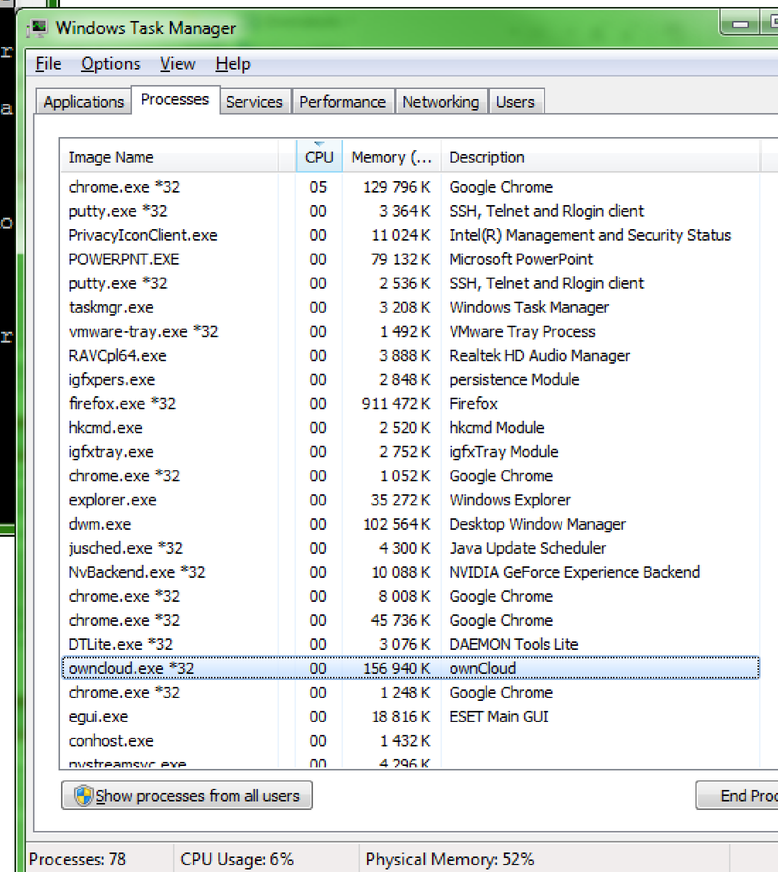
- Activity – task assignment and process vs. thread decision
  - Does the activity need to be multi-programmed?
  - How many parallel execution units are required?
  - How often?
  - Are threads supported in the given system? (see embedded OS-s)

- Pro-s and con-s of the threads
  - Low resource requirement (fast creation)
  - Inside the process: simple (and fast, no overhead) communication with other threads
    - Due to the shared memory
    - The programmer has to design the operation carefully
    - It may lead to errors (see later lecture)
  - Not every platform supports it (most of them does)
  - Communication with threads of another process still complex

- Pro-s and con-s processes
  - The kernel protects the memory range of the process
  - Available on almost every platform
  - Higher overhead
  - The communication with other process are more complex -> higher overhead

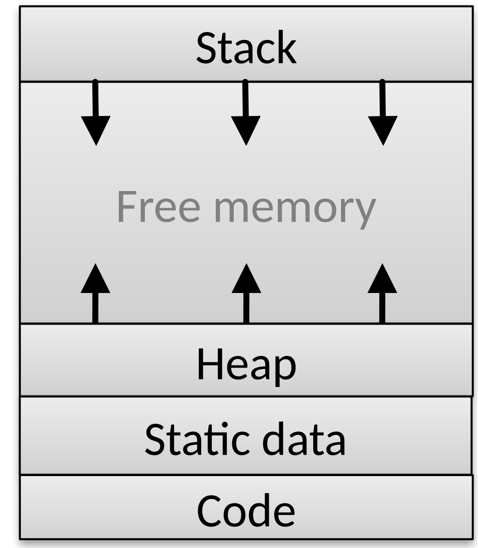
Task managers

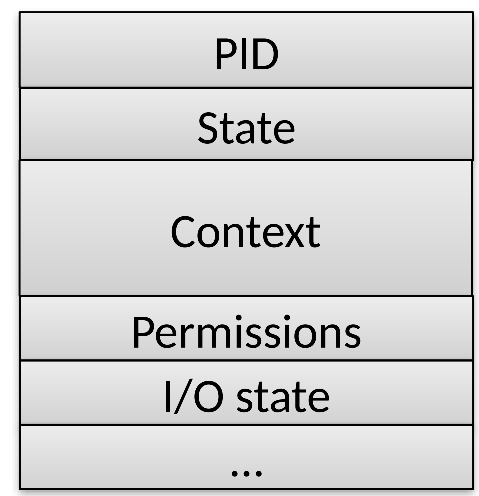




Data structures of the tasks

- Activities performed by programs
  - Tasks have state and life-cycle
  - Tasks have own and administrative data structures
- **Program data** (in the task's memory range)
  - Code
  - Static allocated data
  - Stack: temporary storage, e.g. for function calls
  - Heap: runtime (dynamic) allocated memory space
- **Administrative data** (managed by the kernel)
  - Task (process, thread) descriptor
  - Unique ID (PID, TID)
  - State
  - Context of the task: the descriptor of the execution state
    - Program counter, CPU registers
    - Scheduling information
    - Memory management state
  - Owner and permissions
  - I/O state information





Where to store the administrative data?

- In the kernel's memory range?
  - „Expensive" area, the kernel's memory usage should be minimized
- In the memory range of the process?
  - More difficult to be accessed by the kernel
- How often this data is accessed?
  - Often -> should be stored in the kernel's space
  - Rare -> should be stored in the process' space

- Classification of administrative data
  - Mostly needed when the process is running
    - Permissions
    - State and data of system calls
    - I/O operation data
    - Accounting and statistical data
  - Mostly needed for handling processes
    - ID-s
    - Running and scheduling states
    - Memory management data

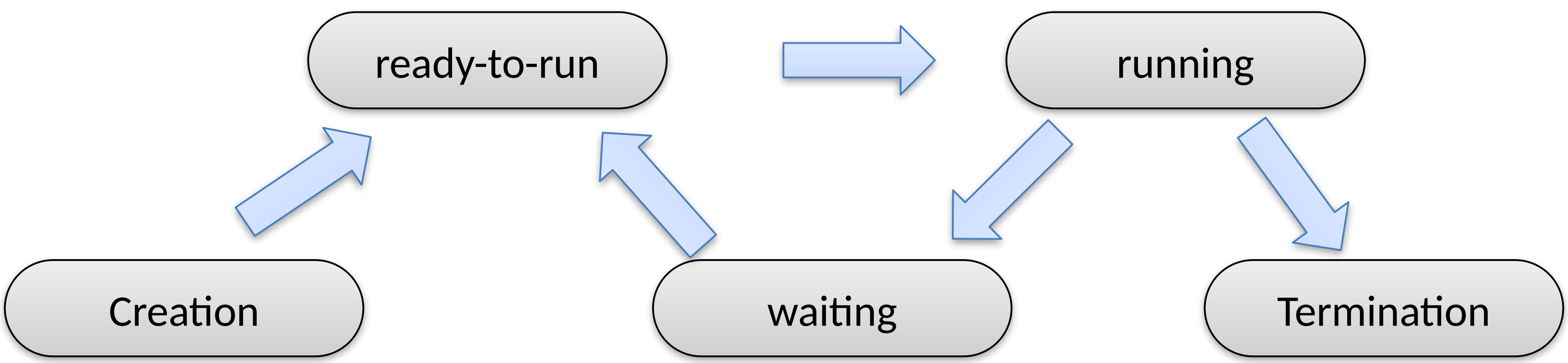
UNIX example

u-space
process-space

proc structure
kernel-space

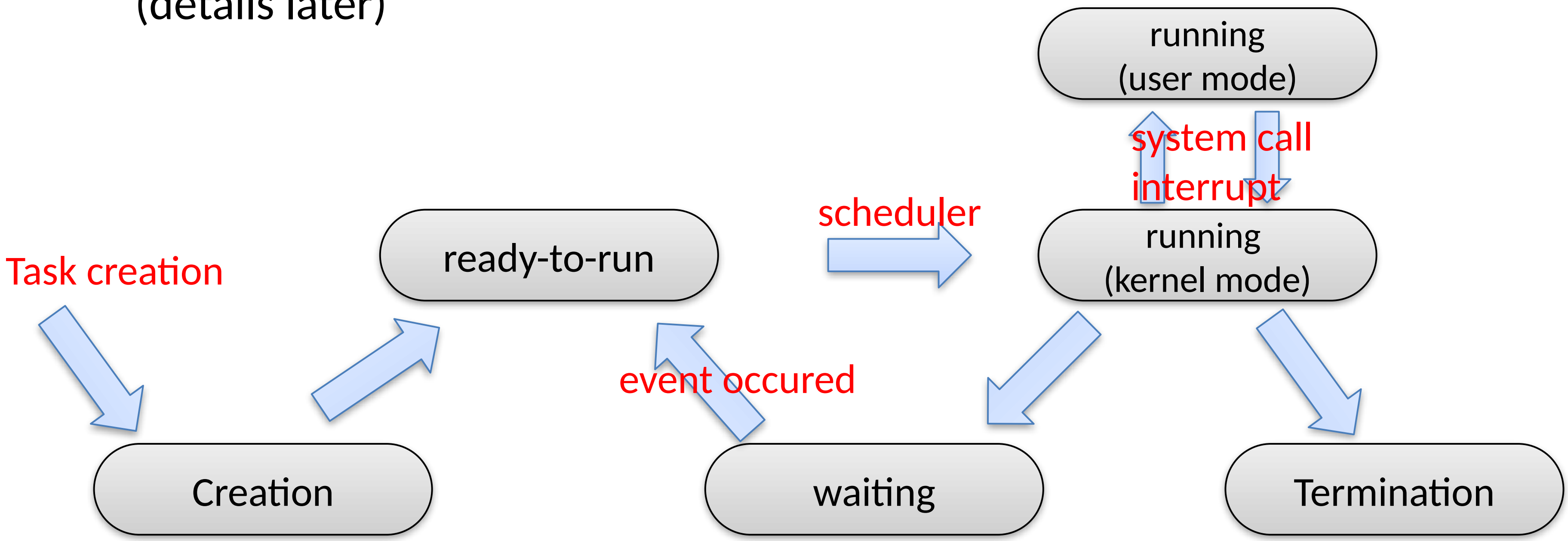
The states of the tasks

- Creation
  - The task's program loaded
  - The kernel creates the data structures and register the new task
  - The task enters into the **ready-to-run** state
- Operation
  - **ready-to-run** (waiting for the CPU)
  - **run** (the task's program is running on the CPU)
  - **waiting** (waiting for a certain event)
- Termination
  - The program terminates itself, or the OS detects a fatal error and terminates the task



State transitions of the tasks

- State transitions are caused by system calls and interrupts
  - The system call also results an interrupt
  - Therefore the state transitions are caused by interrupts
  - Therefore the **kernels are interrupt (event) driven**
- Changing into kernel mode can occurred when the task is in running state
  - The running state can be subdivided (user and kernel mode)
- The transition run –> ready-to-run is performed by the kernel's scheduler (details later)



How are tasks created?

- The first few tasks are created by the kernel when the system boots
- The init or Wininit starts the services of the OS
  - Before the user login, already ~100 tasks are running
- User logs in, and starts programs
- Simple example in UNIX:

```
if ((res = fork()) == 0) {      // child’s branch
   exec(...);      // for example: another program is loaded
     // if returns: exec error
} else if ( res < 0 ) {   // parent’s branch, checking errors
     // for example: if there is any errors during fork()
}
// res = CHILD_PID (>0), the parent’s code runs forth
```

  - The `fork()` method duplicates the current process (starting a new process)
    - All process data is „copied”
  - The `exec()` method loads the new programs code into the initiator programs memory space

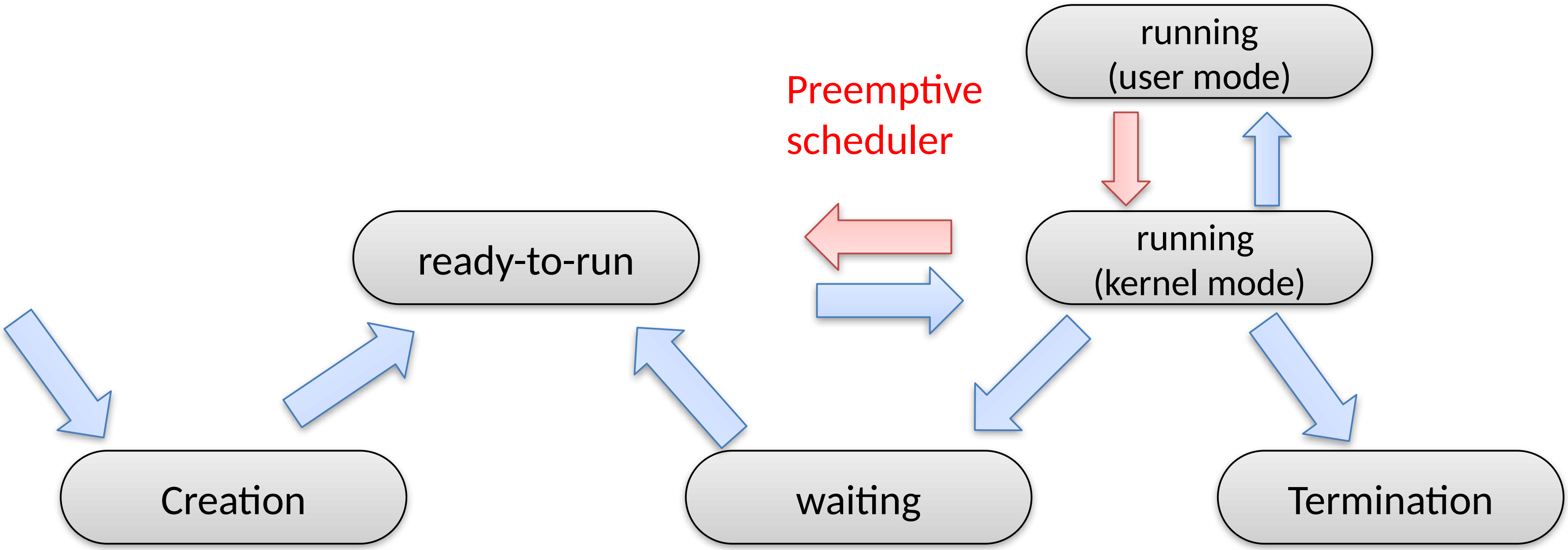
Tree of UNIX processes

- A process can only be created by another process
  - Every process has a parent and may have children
  - In this way the processes can be ordered in a tree
  - The parent can change (if the parent process terminates)
- The fork() method returns the children's PID to the parent
  - The parent can manage its children
- The root process (PID=1, e.g.: init)
  - Parent of every process
  - Runs till the system runs
  - Inherits the „orphan" processes
  - Manages/controls some of the system services
- Family is important
  - The parent gets notification if the child process is terminated

Switching tasks on the CPU

- The running task gives up the right of running (voluntarily)
  - Terminates itself (exit())
  - Performs a system call and waits for its result
- The right of running is taken away from the running process
  - E.g.: time division systems, the process time slice is over
  - The scheduler can take away the right of running in certain systems
  - Due to interrupt or exception (error handling)
- Preemptive and non-preemptive schedulers
  - The **preemptive** scheduler can take away the right of running from the processes
  - When using **non-preemptive** scheduler only the process can give up the right of running
  - The right of running can be taken away in both cases when interrupt or exception (error) occurs

State transitions with preemptive scheduler



The context change

- Context (the descriptor of the execution's state)
  - Program counter (PC), CPU, MMU states, etc.
  - The kernel has its own context, on the level of the kernels own tasks
- If two tasks switching between the CPU, the context has to changed
  - The context of the running task has to be saved
  - The execution state of the former running task has to be restored
  - The control is passed to the now running task
- The interrupts causes context changes (task -> kernel)
  - A small part of the actual context is saved by HW instructions
  - (The interrupt handler performs additional state saving)
  - The interrupt handler runs and returns to point before the IT
  - During the return, the former context is restored
- System calls are works with interrupts -> causing context changes
  - Switching between user and kernel mode is also a context change
- **There are many context changes during the operation of the OS**
  - Context changes should be implemented with minimal overhead
  - In some cases saving the whole context isn't necessary -> IT handler don't change the whole context, only a small part of it (PC, CPU registers…)

Execution mode and context

	User mode	Kernel mode
Task context	The task's program is running	The task is performing a system call
Kernel context	(empty)	IT handling and system management

Summary

- High number of tasks with different nature (simultaneously)
  - I/O intensive (less computation, lot of waiting)
  - CPU intensive (more computation, less waiting)
  - Tasks requiring real-time operation (deadline)
  - Multimedia tasks
  - (There are some system task along user tasks)
  - The user expectations can be various
    - Waiting time, response time, turnaround time, throughput, resource utilization
- The basics of task management
  - Task: a program during execution, it has a state and life-cycle
  - Abstract virtual machine: „virtual" CPU and memory for the tasks
  - Process: a task with its individual memory range, may contain threads
  - Thread: A task with sequential operation, it may share memory with other threads
- The life-cycle of tasks
  - Creation, ready-to-run, run, waiting, termination
  - The context changes are caused by interrupts
  - The task change means context change, which is often during the kernel's (and the OS) operation