Operating Systems

Lecture 3 - Scheduling

Adrien Krähenbühl

Master of Computer Science
PUF - Hồ Chí Minh

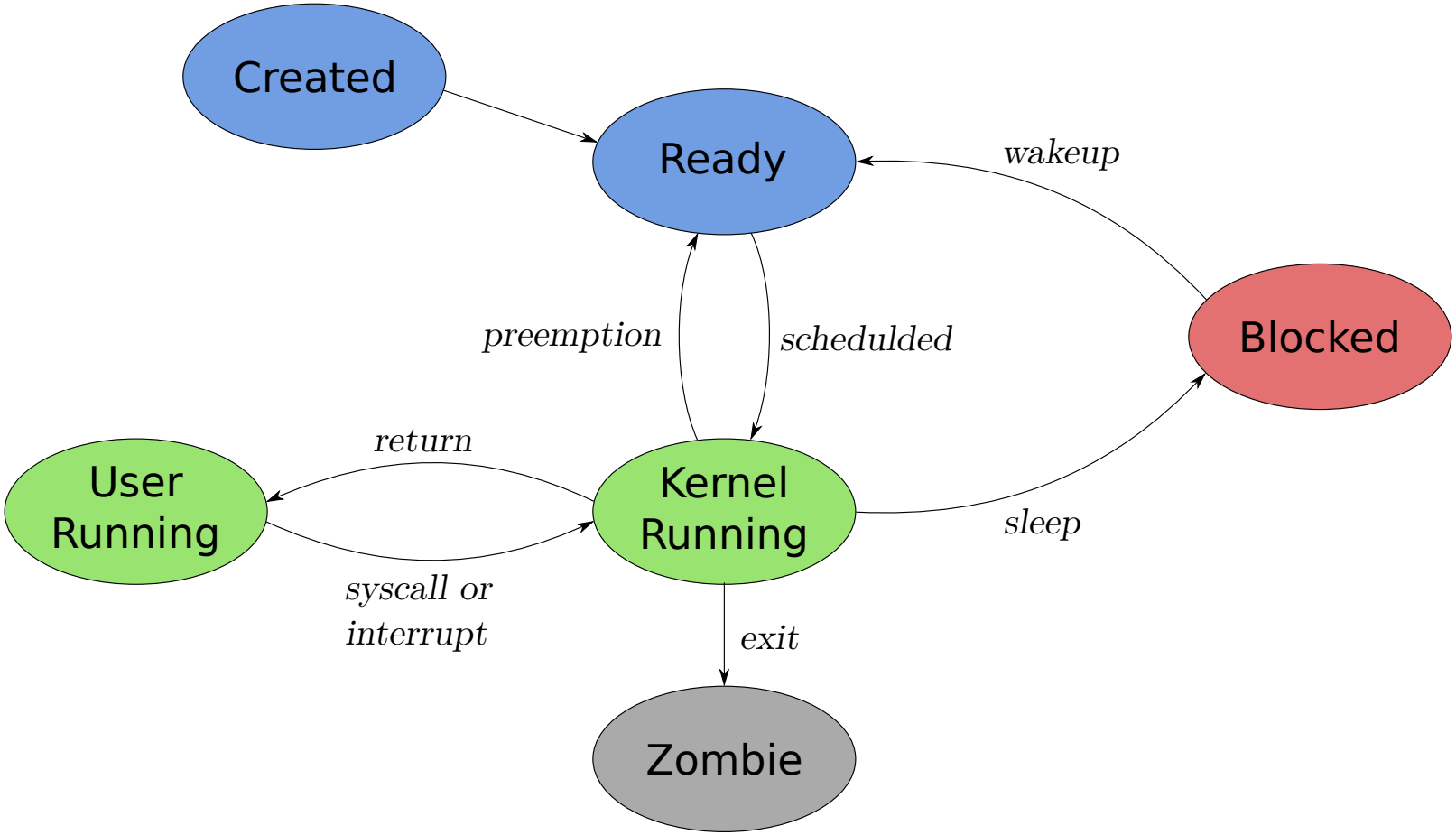
2016/2017

Preemption & signals

Scheduling algorithms

System execution

Process states



Scheduler execution

- ✓ **The kernel doesn't exist**
  - No "man in black" to schedule tasks in background
- ✓ The scheduling requires execution of the **scheduler code**:
  - on the good processor
  - in kernel mode
- ✓ Implicit or explicit call (or forced)

Cooperative scheduling

- ✓ Explicit scheduling by applications
  - a process that handing over the processor (`sched_yield()`)
  - a process that is waiting (`sleep()`)
- ✓ Implicit scheduling
  - Blocking system call with `read()`, `poll()`, ...
  - Return of the system call requires the end of the I/O
- ✓ Any guarantee of regular scheduling: depends on applications
- ✓ Equity and reactivity problem
  - A processor can kept the processor indefinitely
  - Tasks must cooperate

Scheduler execution

Need a task in kernel mode:

- ✓ on the good processor
- ✓ it must call the scheduler code

How to schedule from the code of an application that doesn't make explicit or implicit scheduling?

- ✓ Each time that the system hands over the processor
- ✓ Helped by system calls and interruptions

Preemption

Schedule a task by force:

- ✓ If a task is executed for too long
  - move it at the end of the run queue
  - replace it by the first in the run queue
- ✓ If another one has priority:
  - move the current task in run queue: at the end or beginning?
  - replace it by the first priority task $\rightarrow$ risk of starvation?
- ✓ Preemption has a small overhead
  - require to check sometimes if a forced scheduling is required
- ✓ But huge gain in equity and reactivity
  - No more need for process cooperation!

Preemption in Linux

- ✓ The scheduler checks **regularly** whether to hands over another task
  - During a switch into user space, during an interruption or at the end of a system call
  - Usefulness of periodic clock tics
- ✓ We only mentioned preemption for user code: the kernel code can be long and affect the system reactivity
  - Linux 2.6 preempt also the kernel code

Who preempt who?

- ✓ No direct preemption in user space
  - execution of scheduler code can only be done in kernel space
  - can happen anywhere in user code through interruptions
- ✓ Processes preempted in kernel mode by the same way:
  - For example just before the switch into user mode (after system calls, exceptions, interruptions, . . .)
- ✓ Interruptions can preempt at anytime (except if deactivated)

Signals

Signal

Message (with a number) from a process to another one.

- ✓ A signal is a **interruption from applications**
- ✓ Received by execution of an handler
  - Routing of the current program then resuming
- ✓ Delivered in user space
  - Processed by the scheduler as preemption
  - Only if there is system calls or interruptions
  - Hopefully, there is the clock tics

Signals

The kernel modifies the execution context of the process before let it to be executed

- ✓ Force execution of the handler
- ✓ Force execution of a function "handler return" at the end of the handler
- ✓ The "handler return" replaces the initial execution context
  - execution resuming at the initial location

Preemption & signals

Scheduling algorithms

System execution

Needs

- ✓ Progression of all tasks with equity and interactivity
- ✓ Any dependency of task's choices
  - preemption

Scheduling time

Time slice

A time slice is a quantum of execution time

If a time slice is:

Too short to increase the overhead % of context switching

Too long to reduce interactivity:

- ✓ tasks are allowed to run longer before a context switch is forced
- ✓ amortizes overhead of context switch

In Linux:

- ✓ Depends on the priority:
  - 100ms by default
  - From 5ms to 800ms from the lowest to the highest priority

Measure of scheduling time

- ✓ Hard to precisely know if a process requires a lot of processor time
- ✓ Scheduling mainly based on clock ticks
  - Not really precise: ~4ms in Linux/x86
  - Use the cycle counter if exists
- ✓ How count the interruption time?

SMP and NUMA

Preemption Concurrency between tasks on a same processor

→ Preventable by deactivation of interruptions

Multiprocessor Real concurrency between tasks

- ✓ Anytime
- ✓ Critical synchronization!
- ✓ What task on which processor?

SMP and NUMA

- ✓ Affinity between processes, threads, processors and/or memory
  - Notion of preference in scheduler
- ✓ Required to keep a trace of affinities
  - Hard: create one queue per core and keep the process inside at the end of its time slice?
  - Compromise with load balancing
- ✓ When break the affinity?
  - If one is free and another overloaded
  - Several strategies

Scheduling criterion for the user

- ✓ Performance
  - Speed of execution or termination
  - Reactivity
  - Deadlines
- ✓ Equity?
- ✓ Predictability?

Scheduling criterion for the system

- ✓ Performance
  - Work quantity
  - Processor use (and devices)
- ✓ Equity
- ✓ Respect of priorities

Scheduling types

- ✓ FIFO (First In First Out)
- ✓ Round-Robin
  - turn-based execution during time slices
- ✓ Short First or Short Remaining Next
  - Requires to plan the execution duration
- ✓ Feedback
  - Penalty for greedy processes (adjustment of initial priority)

Scheduling for multiprocessors

- ✓ Only one run queue (Load Sharing)
  - All processors take it in.
  - Contention problem if few processors
- ✓ One run queue by processor
  - No contention
  - Need of load balancing

Special scheduling for multiprocessors

- ✓ By group
  - Interesting if cache/memory affinity
  - Interesting if inter-task communications
- ✓ Dedicate a processor for a task
  - Others tasks cannot use the processor
  - 100% of CPU time
  - Guarantee of cache/memory affinity
  - Not be confused with the processor assignation: free processor but constrained task

Priorities in Unix

- ✓ Dynamic priority (Feedback strategy)
- ✓ Factor can be set by the user (nice)
  - need to be privileged to adjust it
- ✓ Very priority swapping
  - Globally, priority increases from applications to devices

Adapt scheduling to priorities

- ✓ Execute priority tasks before the others?
  - Not really useful, except for very short processes
- ✓ Execute priority tasks more often
- ✓ Execute priority tasks longer

Queue sorted by priority

- ✓ Linear complexity, in the number of tasks
- ✓ Each scheduling operation has an expensive cost : `yield`, creation, destruction
- ✓ Be careful to starvations
  - Non priority tasks must not stay indefinitely at the end of queue
  - Priority adjustment depends on previous privileges? (in constant time?)

Queues by priority

- ✓ A set of queues of tasks of same priority
  - Use in constant time
- ✓ Queues parsed more or less often according to the priority
- ✓ Limited number of queues:
  - 40 in Linux
  - Possibility to use skip-lists

Adjust time slice according to priority

- ✓ Define a quantum of execution time depending on priorities
  - Potentially dynamic
- ✓ Good if interactive: I/Os can put to sleep before the end of time slice
- ✓ Bad if too long in processor: preempted when the time slice ending
- ✓ Can be computed in constant time

Scheduling in Linux

- ✓ Unix scheduling is basic but evolve quickly
- ✓ Good interactivity and equity
- ✓ Effective priority with bonus or penalty:
  - interactivity credit if the process is sleeping
  - priorities sometimes only respected inside sessions
- ✓ Support a lot of processes and processors
  - Advanced optimizations of data structures to be scalable
  - Mask of accepted (and/or preferred) processors for each process
- ✓ A lot of small optimizations
  - Ex: `vfork()` child preempts its parent to avoid a copy-on-write

Scheduling complexity

- ✓ Costs of operations can't depend on task number
- ✓ All the time a lot of blocked tasks that must not be in the run queue
- ✓ Potentially a lot of Ready tasks
  - never iterate over all the queue
- ✓ Insertion of a not scheduled task: do not sort the queue
  - How to manage priorities?
- ✓ Remove a scheduled task: do not iterate over all the queue
  - How quickly choose the task to execute?
- ✓ Time slice computing
  - do not iterate over the queue to compute the sum of task weights
  - maintain the sum at each task scheduling

Case of real time constraints

- ✓ Reactivity
- ✓ Priority
- ✓ Deadline (execution before a date)
- ✓ Use a special scheduling case:
  - Time slice can be infinite (require privilege)
  - `SCHED_RR` and `SHED_FIFO` in Linux
  - see manpage of `sched_setscheduler()`

Real time scheduler

- ✓ Non general scheduler: dedicated to real time
- ✓ Maximize the reactivity
  - Minimize deactivation of interruptions
  - Minimize locks
- ✓ Routine dedicated to management of real time constraints
  - delay, pause, alarm (Timer), ...

Preemption & signals

Scheduling algorithms

System execution

The kernel does not exist

- ✓ Any kernel process monitoring everythings
- ✓ Any “man in black” taking the processes, executing them, checking what they do, . . .
- ✓ The system is mainly executed in the process context, when they are in kernel mode
- ✓ Just eventually daemons dedicated to periodic tasks

Kernel execution in process background

The kernel is executed:

- ✓ through "hooks", in non-dedicated code
  - Ex: scheduling at return of `getpid()`
- ✓ during system calls
  - Processing by system calls
  - Additional processing at the same time
- ✓ during interruptions/exceptions
  - Immediate processing required
  - Deferred processing if necessary: expensive tasks could affect the reactivity

Execution in kernel mode

Context switch:

- ✓ privilege switch
- ✓ use of a special stack:
  - allows to avoid storage of critical informations in user space
  - user and kernel memory spaces do not have the same constraints
  - Ex: No page default in kernel mode with Linux
- ✓ Saving and restoration of user registers

Case of micro kernels

- ✓ Minimal system calls
  - messages passing between components in user space
- ✓ Effective OS management outside the kernel
  - In the dedicated server processes

Kernel threads

- ✓ Monolithic kernels need additional processing
  - Unpredictable system calls
  - Interruption handlers in too restricted context
- ✓ Use of dedicated threads
  - All the time in kernel mode (never in user mode)
  - Periodic and/or planned tasks

Kernel threads are everywhere

Any process is launched by a kernel thread

- ✓ they switch into user mode just after their creation
- ✓ they come back in kernel mode:
  - during a system call
  - just before it was killed

A multithreaded process is launched by one or many kernel threads according to the 1:1, 1-M or N-M model.

How to know who I am?

The kernel often needs to know who it is

- ✓ All processes may be required to run most of the kernel code
- ✓ Kernel threads too, see the interruption handlers

How to know which task is executed?

- ✓ What rights does it have? User, permissions, ...
- ✓ Which resource does it have? virtual memory, ...

Used a register?

- ✓ We already have a register to locate the stack

Linux use a nice hack:

- ✓ The task descriptor is placed under the stack
- ✓ Task = truncated stack address
- ✓ current = stackPointer & ~`STACK_SIZE`
- ✓ Requires to align and limit stacks