NTNU | Norwegian University of Science and Technology

Operating Systems

Lecture 12: Uniprocessor scheduling

Michael Engel

Processes once again…

- Processes are (still…) the central abstraction for activities in current operating system
  - illusion of independent sequential control flows as a concept (sequence of CPU and I/O bursts)
  - in real life, the CPU is *multiplexed*
- Unix systems provide a set of system calls to create and manage processes and to provide communication channels
  - in addition, modern operating systems also support light- and featherweight processes
- Processes are controlled by the operating system
  - allocation of resources
  - preemption of resources

Dispatch states

Depending on the scheduling level, every process is assigned a ***logical state*** representing its ***dispatch state*** at a given point in time:

- ***short-term scheduling***
  - ready, running, blocked
- ***medium-term scheduling***
  - swapped and ready, swapped and blocked
- ***long-term scheduling***
  - created, terminated

Rule of thumb how often a scheduling decision or state change occurs:
- **short term**: μs – ms
- **medium term**: ms – min
- **long term**: min – hours

Short-term scheduling

- **ready** to be executed by the CPU
  - a process is on the *ready (waiting) list* for CPU allocation
  - its list position depends on the scheduling algorithm

- **running**: resource "CPU" has been allocated to the process
  - a process is computing: "CPU burst"
  - there is *only one running process per CPU* at any given moment in time

- **blocked**: waiting for an event
  - a process performs input or output: "I/O burst"
  - it waits for the occurrence of at least one condition

Medium-term scheduling

A process is completely swapped out

- the complete contents of its address space are moved to background storage
- the main memory it used is released

The process has to wait to be swapped in:

- swapped out ready (READY SUSPEND)
    - CPU allocation ignores this process
    - the process is on a waiting list for memory allocation
- swapped out blocked (BLOCKED SUSPEND)
    - the process waits for an event (it is blocked)
    - if this event takes place, the process state changes to READY SUSPEND

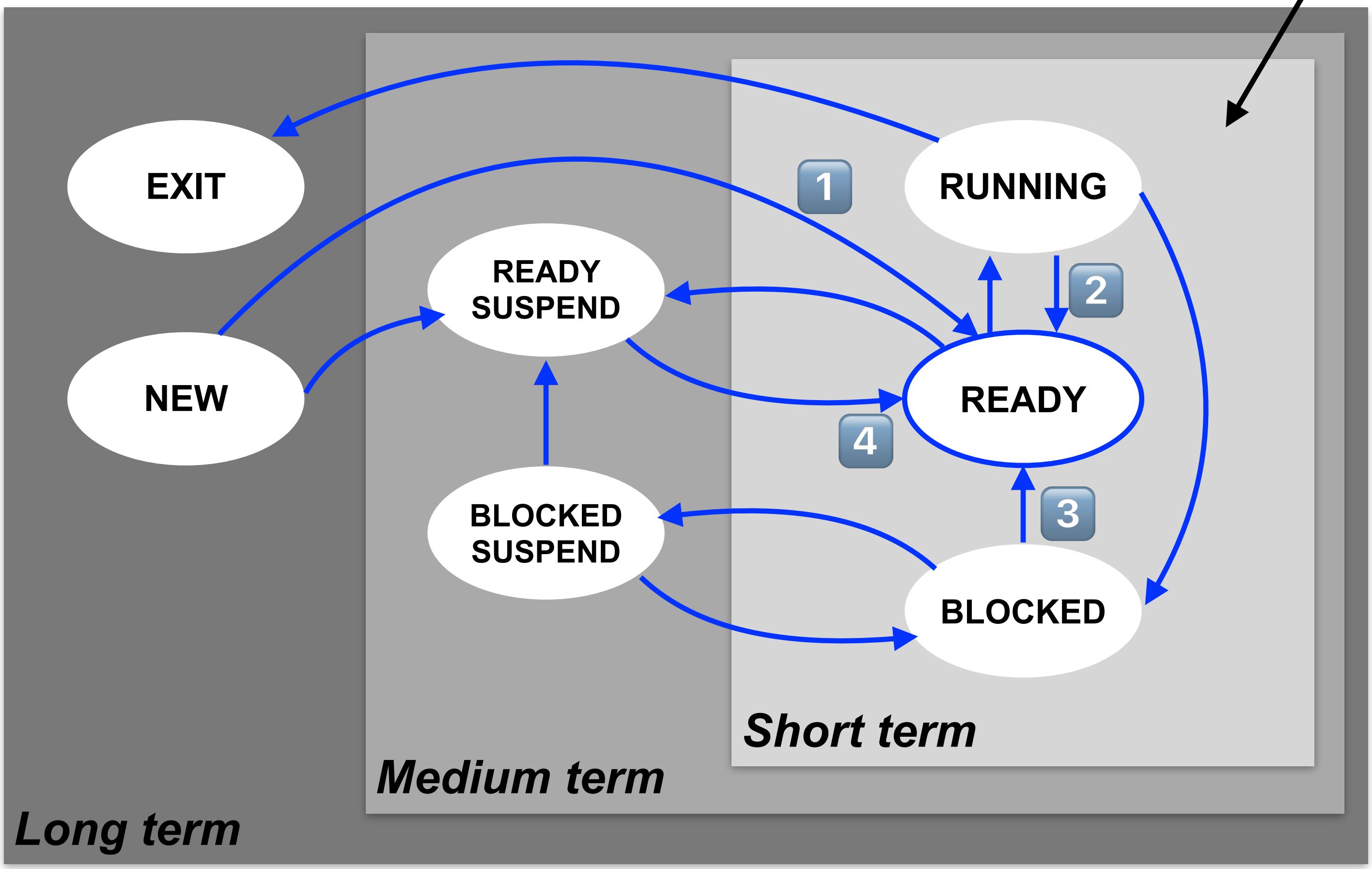
Long-term scheduling

- Processes are ***created*** (NEW) and ready to be started:
  `fork(2)`
  - a process instance was created and assigned to a program
  - the allocation of the resource "memory" might still be outstanding (e.g. when paging in parts of the process address space on demand)

- Processes are ***terminated*** (EXIT) and wait for their removal:
  `exit(2)/wait(2)`
  - the process is terminated, its resources are released
  - the "cleanup" after process termination can be performed by a different process (e.g. in Unix)

State transitions

We focus on **short term scheduling** now



Scheduling points

- Every transition into the READY state updates the CPU waiting queue
  - a decision about the queueing of its process control blocks is made
  - the result depends on the CPU allocation strategy of the system
- ***Scheduling*** and ***rescheduling*** takes places…
  1. after a process is created
  2. if a process yields control of the CPU
  3. if the event a process is waiting for takes place
  4. when a swapped out process is considered for CPU allocation again
- A process can be forced to yield (release) the CPU
  → ***preemptive scheduling***
  - e.g. using a timer interrupt

First-Come First-Served – FCFS

Classical scheduling

- A simple and fair (?) algorithm: "first come first served"
- Queueing criterion is the *arrival time* of a process
- Algorithm is *non preempting* and assumes *cooperating processes*

Process	Times					
	arrival	service time T_s	start	end	runtime T_r	T_r/T_s
A	0	1	0	1	0	1.00
B	1	100	1	101	100	1.00
C	2	1	101	102	100	**100.00**
D	3	100	102	202	199	1.99
average						26.00

- Example:
  - the normalized runtime (T_r / T_s) of C is bad in relation to its service time T_s

Discussion: FCFS – "convoi effect"

- This problem affects short running **I/O-intensive processes** which follow long **CPU-intensive processes**
  - Processes with **long** CPU bursts benefit from this
  - Processes with **short** CPU bursts are disadvantaged
- FCFS minimizes the number of *context switches.* However, the **convoi effect** causes a number of problems:
  - large response time
  - low I/O throughput
- If the system runs a mix of CPU- and I/O-intensive processes, FCFS is not a suitable approach
  - it is typically only used in *batch processing* systems

Round Robin (RR)

Classical scheduling

- Reduces the disadvantage of processes with short CPU bursts: "everyone for themselves!"
  - the available processor time is split into ***time slices***
- When a time slice is used up, a process switch ***can*** occur
  - the interrupted process is moved to the end of the ready list
  - the next process is selected from ready list according to FCFS
- Basis for protecting access to the CPU:
  a timer enforces an interrupt at the end of each time slice
- The efficiency of this approach depends essentially on the chosen length of the time slice
  - too long ➛ round robin degenerates to FCFS
  - too short ➛ very high overhead for process switches
- Rule of thumb: time slices should be "a bit longer" than the duration of a "typical interaction"

Discussion: RR – performance problems

- I/O-intensive processes terminate their CPU burst before their time slice is used up
  - they block and are added back to the ready list when their I/O burst is finished
- CPU-intensive processes, however, use their time slice completely
  - they are then preempted and immediately added to the end of the ready list
- The amount of CPU time for processes is thus distributed inequally ➢ CPU-intensive processes get a larger share
  - I/O-intensive processes are not served as well, thus the utilization of I/O devices is low
  - the *variance of the response time* of I/O-intensive processes increases

Virtual Round Robin (VRR)

Classical scheduling

- Avoids the unequal distribution of CPU times with RR
  - processes are added to a *preferred list* when their I/O burst ends
  - this list is considered before the ready list

- Virtual Round Robin uses time slices of *different lengths*
  - processes on the *preferred list* are only allocated a *partial time slice*
  - they can use the *remaining run time* they did not use in their previous time slice
  - if their CPU burst last longer, they are moved to the ready list
- Scheduling in VRR involves a bit more overhead compared to RR

Shortest process next (SPN)

Classical scheduling

- Reduces the disadvantage of short CPU bursts with FCFS: "let the shortest come first…"
  - this requires knowledge about the process run times
  - no preemption
- The main problem here is the ***prediction of run times***
  - batch processing: the programmer annotates the required *time limit*
  - interactive procession: time limit estimated based on previous CPU burst lengths of the process
- Response times are reduced significantly and the overall system performance is increased
  - However: danger of ***starvation*** of CPU-intensive processes

Discussion: SPN – weighting bursts

- CPU bursts further in the past should be weighted less:

$$S_{n+1} = \alpha \cdot T_n + (1-\alpha) \cdot S_n$$

  - values of the constant weighting factor α: $0 < \alpha < 1$
  - it represents the *relative weighting* of single CPU bursts in the time line of the process

This ***statistical approach*** is also called **exponential smooting**

- Recursive solving leads us to...

$$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots + (1-\alpha)^i \alpha T_{n-i} + \ldots + (1-\alpha)^n S_1$$

$$S_{n+1} = \alpha \cdot \sum_{i=0}^{n-1} (1-\alpha)^i T_{n-i} + (1-\alpha)^n S_1$$

- for α = 0.8:

$$S_{n+1} = 0.8\mathrm{T}_n + 0.16\mathrm{T}_{n-1} + 0.032\mathrm{T}_{n-2} + 0.0064\mathrm{T}_{n-3} + \ldots$$

Shortest Remaining Time First (SRTF)

Classical scheduling

- Extends SPN with preemption
  - thus appropriate for interactive operation
  - results in improved runtimes
- The running process is preempted if $T_{exp} < T_{rest}$
  - T_{exp} is the *expected* CPU burst length of an arriving process
  - T_{rest} is the *remaining* CPU burst length of the running process
- Difference to RR:
  SRTF is ***not*** based on timer interrupts, but nevertheless preemptive
  - We have to estimate burst lengths instead
- Like SPN, processes can also starve using SRTF

Highest Response Ratio Next – HRRN

Classical scheduling

- Avoids the possible starvation of CPU-intensive processes that can occur with SRTF
  - HRRN considers the ***aging*** of processes – their *waiting time*

$$R = \frac{w + s}{s}$$

  - *w* is the *"waiting time"* the process has accumulated so far
  - *s* is the "expected ***service time***"

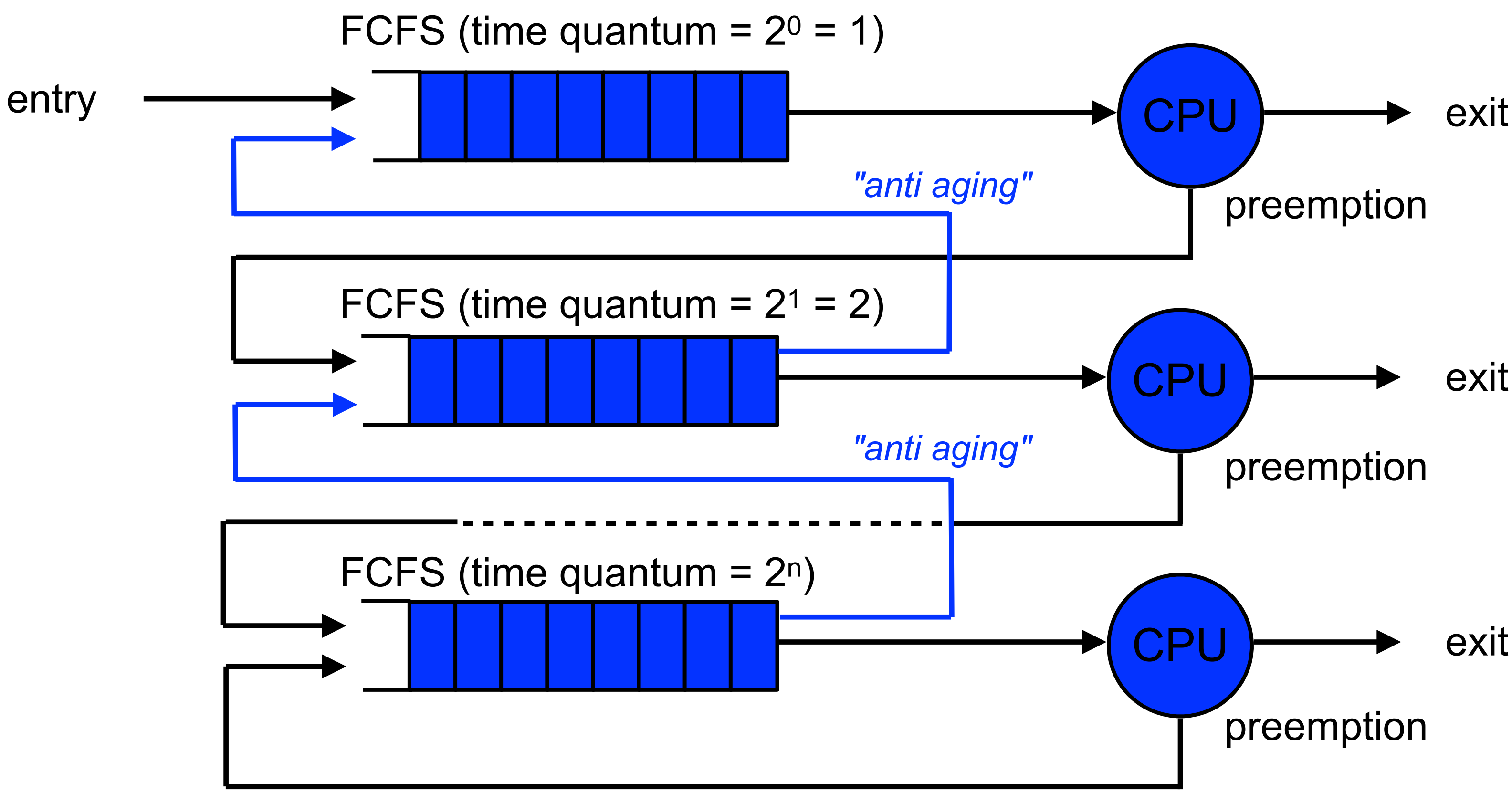
- HRRN always selects the process with the highest value of *R*
  - Again, this is based on an ***estimation*** of the ***service time***

Feedback (FB)

Classical scheduling

- Short processes obtain an advantage without having to estimate the relative lengths of processes
  - Basis is the ***penalization*** of long running processes
  - Processes are preempted
- Multiple ready lists used according to number of priority levels
  - when a process arrives for the first time, it has highest priority
  - when its time slice is used up, it is moved to the next lower priority level
  - the lowest level works according to RR
- Short processes finish in a relatively short amount of time, but long processes can starve
  - It is possible to consider the waiting time to move a process back to a higher priority level (**anti-aging**)

Feedback (FB) scheduling model



"Multilevel feedback queues"

Discussion: Priorities

- Process priorities significantly influence scheduling decisions
- **Static priorities** are defined when a process is created
  - their value cannot be changed during the execution of the process
  - this enforces a *deterministic ordering* of processes
- **Dynamic priorities** are updated while a process is running
  - the operating system usually updates the priorities, but also the user can be allowed to influence priorities
  - SPN, SRTF, HRRN and FB are special cases of this approach

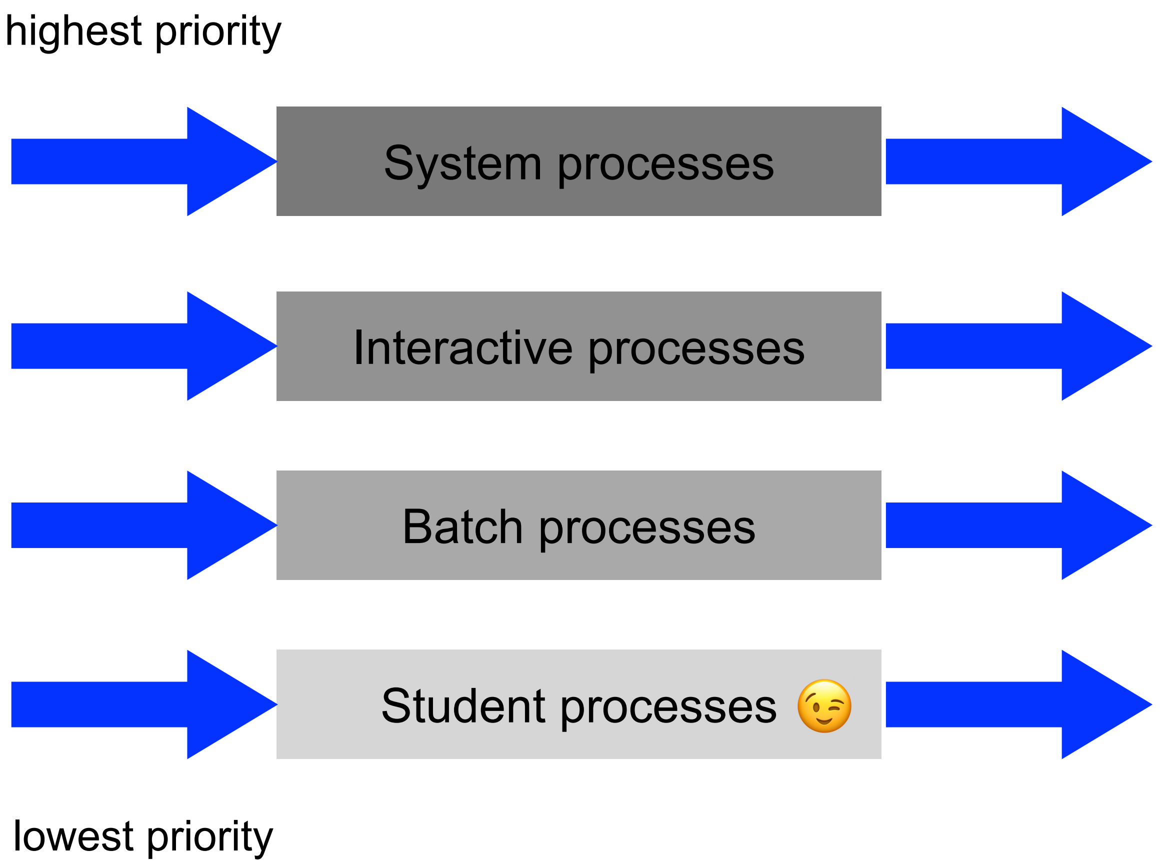
Combination – Multi-level scheduling

- Multiple scheduling strategies can be combined (i.e., used "simultaneously"), e.g. support of
  - interactive and background processing or
  - realtime and non-realtime processing
    - interactive / real-time critical processes are preferred

- The implementation typically uses multiple ready lists
  - every ready lists has its own scheduling strategy
  - the lists are typically processed using priority, FCFS or RR
  - overall, a very complex approach!

- FB can be seen as a special case of this approach

Combination – Multi-level scheduling



(adapted from Silberschatz)

Objectives for evaluation

- User oriented:
  - **Run time** – time between start and termination of a process *including the waiting time(s)* → **batch processing**
  - **Response time** – time between user input and program response → **interactive systems**
  - **Tardiness** – for the interaction with external physical processes, deadlines have to be adhered to → **real-time systems**
  - **Predictability** – processes are always processed identically independent of the load → **hard real-time systems**
- System oriented:
  - **Throughput** – finish as many processes as possible per time unit
  - **CPU load** – keep the CPU busy at all times
    - avoid overhead (scheduling decisions, context switches)
  - **Fairness** – no process should be disadavantaged (e.g. by *starvation*)
  - **Load balancing** – I/O devices should also be utilized uniformly

Quantitative comparison

	Process Start Service time T_s	A 0 3	B 2 6	C 4 4	D 6 5	E 8 2	average
FCFS	End Runtime T_r T_r/T_s	3 3 1.00	9 7 1.17	13 9 2.25	18 12 2.40	20 12 6.00	8.60 2.56
RR q=1	End Runtime T_r T_r/T_s	4 4 1.33	18 16 2.67	17 13 3.25	20 14 2.80	15 7 3.50	10.80 2.71
SPN	End Runtime T_r T_r/T_s	3 3 1.00	9 7 1.17	15 11 2.75	20 14 2.80	11 3 1.50	7.60 1.84
SRTF	End Runtime T_r T_r/T_s	3 3 1.00	15 13 2.17	8 4 1.00	20 14 2.80	10 2 1.00	**7.20** **1.59**
HRRN	End Runtime T_r T_r/T_s	3 3 1.00	9 7 1.17	13 9 2.25	20 14 2.80	15 7 3.50	8.00 2.14
FB q=1	End Runtime T_r T_r/T_s	4 4 1.33	20 18 3.00	16 12 3.00	19 13 2.60	11 3 1.50	10.00 2.29

Qualitative comparison

Strategy	preemptive/ cooperative	prediction required?	implement. overhead	starvation possible	effect on processes
FCFS	cooperative	no	minimal	no	convoi effect
RR	preemptive (timer)	no	low	no	fair, but dis-advantage for I/O-int. proc.
SPN	cooperative	yes	large	yes	disadvantage for CPU-int. processes
SRTF	preemptive (at start)	yes	larger	yes	disadvantage for CPU-int. processes
HRRN	cooperative	yes	large	no	good load distribution
FB	preemptive (timer)	no	larger	yes	can prefer I/O-intensive processes

Scheduling in Unix

- Two step preemptive approach
  - objective: reduce response times
- No long term scheduling
- **high-level**: mid term, using swapping
- **low-level**: short term preemptive, MLFB, dynamic process priorities

$$prio = cpu\_usage + p\_nice + base$$

  - Once a second:
    - every "tick" (1/10 s) reduces the "usage entitlement" for the CPU by increasing *cpu_usage* for the running process
    - high *prio* value = low priority!
  - The amount of *cpu_usage* over the time is reduced (*smoothed*)
    - the smoothing function is different in various versions of Unix

UNIX – 4.3 BSD (1)

- The user priority is determined at every fourth tick (40ms):

$$P_{usrpri} = min(PUSER + \frac{P_{cpu}}{4} + 2 \cdot P_{nice}, 127)$$

- P_{cpu} is incremented (by 1) with every tick and is smoothed once a second:

$$P_{cpu} \Leftarrow \frac{2 \cdot load}{2 \cdot load + 1} \cdot P_{cpu} + P_{nice}$$

- Smooting for processes that are woken up and were blocked for more than 1 second:

$$P_{cpu} \Leftarrow \left(\frac{2 \cdot load}{2 \cdot load + 1}\right)^{P_{slptime}} \cdot P_{cpu}$$

UNIX – 4.3 BSD (2)

- Smoothing (using a *decay filter*):
  for an assumed average load of 1: $P_{cpu} := 0.66 \cdot P_{cpu} + P_{nice}$

- In addition, we assume that a process collects T_i ticks in the time interval i and $P_{nice} = 0$

$$P_{cpu1} = 0.66\, T_0$$
$$P_{cpu2} = 0.66\, (T_1 + 0.66\, T_0) = 0.66\, T_1 + 0.44\, T_0$$
$$P_{cpu3} = 0.66\, T_2 + 0.44\, T_1 + 0.30\, T_0$$
$$P_{cpu4} = 0.66\, T_3 + \ldots + 0.20\, T_0$$
$$P_{cpu5} = 0.66\, T_4 + \ldots + 0.13\, T_0$$

- After 5 seconds, only 13% of the "old" load are considered

Windows NT – Priority classes

- Preemptive, priority- and time slice-based *thread scheduling*
  - preemption also occurs for threads executing in the kernel → different to Unix
  - RR for processes of the same priority:
    0 reserved, 1–15 variable, 16-31 real-time
- The *thread type* (fore-/background thread) determines the *time quantum* available to the thread → *quantum stretching*
  - quantum (between 6 and 36) is reduced by 3 or 1 with every tick (10 or 15 ms), if the thread changes to the waiting state
  - the length of a time slice varies with the process: 20–180 ms
    - foreground/background, server or desktop configuration
- In addition, NT has *variable priorities*:
  - process_priority_class + relative_thread_priority + boost

NT – Adaptive priorities

- Thread priorities are *dynamically increased* when certain conditions are given: ***dynamic boost***
  - Completion of input/output (disk): +1
  - Mouse movement, keyboard input: +6
  - Deblocking, release of resources (semaphore, event, mutex) +1
  - Other events (network, pipe, …) +2
  - Event in foreground process +2
- Dynamic boosts are decreased again ("used up") with every tick
- **Guarantee of progress**
  - avoids the *starvation* of threads
  - up to 10 "disadvantaged" threads are allocated priority 15 for two time slices every 3–4 seconds

Conclusions

- Operating systems take CPU scheduling decisions on three different levels:
  - **Long term scheduling**: admission of processes to the system
  - **Medium term scheduling**: swapping of processes
  - **Short term scheduling**: short-term CPU allocation

- All algorithms discussed in this lecture are considered short term scheduling approaches:
  - there are different **user- and system oriented criteria** to assess the properties of a CPU scheduling algorithm
  - the selection of an approach is difficult and can have unexpected negative effects
  - the "best" approach can only be found by an analysis of typical application profiles and all given constraints