Course
TÖL401G: Stýrikerfi / Operating Systems
5. CPU Scheduling

Chapter Objectives

- Introduce CPU-scheduling, which is the basis for multiprogrammed operating systems.
- Introduce scheduling evaluation criteria.
- Describe various CPU-scheduling algorithms.
- Provide a glimpse into real-time scheduling and thread scheduling.
- Explain the issues related to multiprocessor/multicore scheduling and hyper-threading/hardware multithreading.
- Examine the scheduling algorithms of Microsoft Windows.

Contents

Note: The scheduling algorithms in Section 5.3 are presented slightly different than in the Silberschatz book. Furthermore, a section on real-time scheduling has been removed in newer book editions, but I kept one slide.

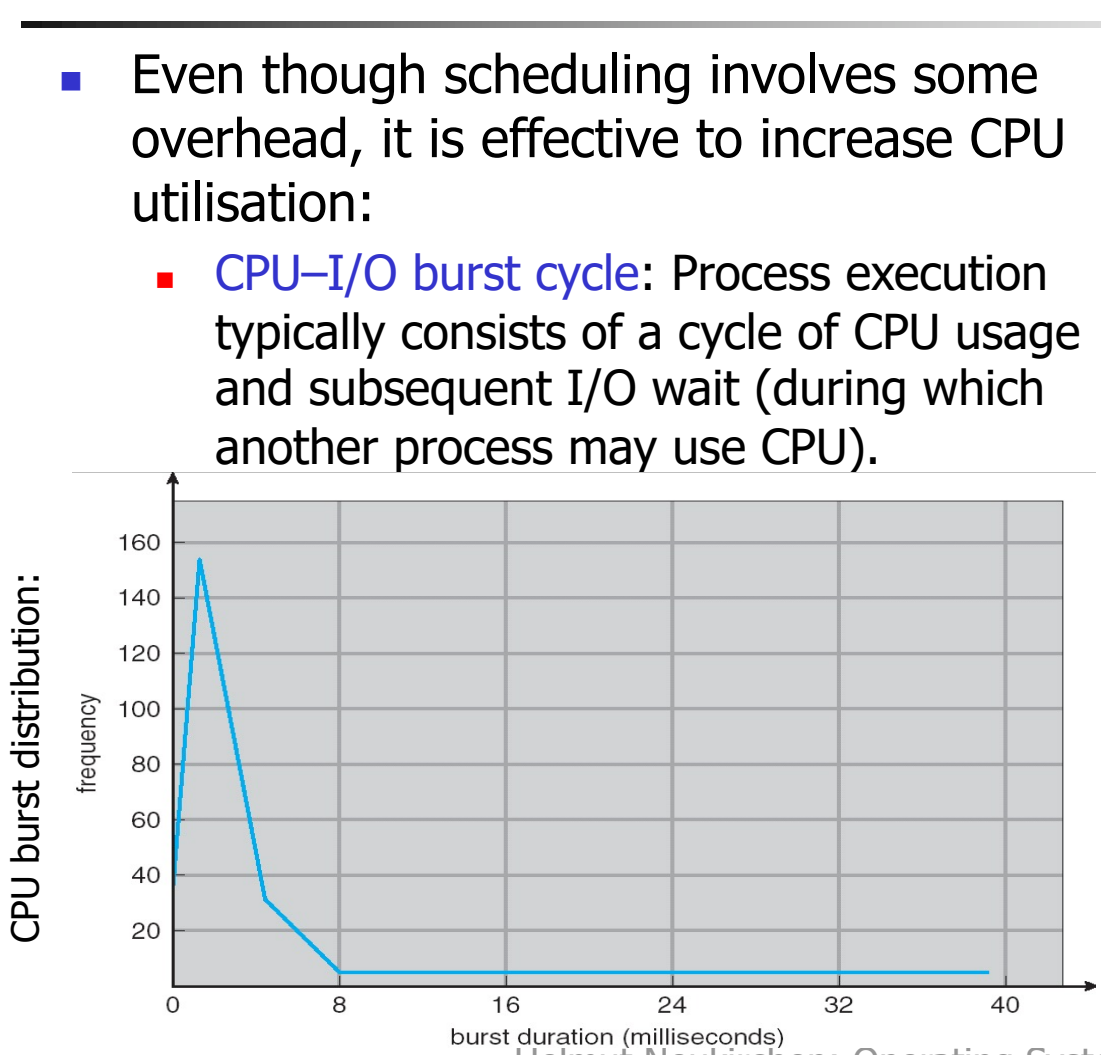
5.1 Basic Concepts of Scheduling: Motivation for scheduling

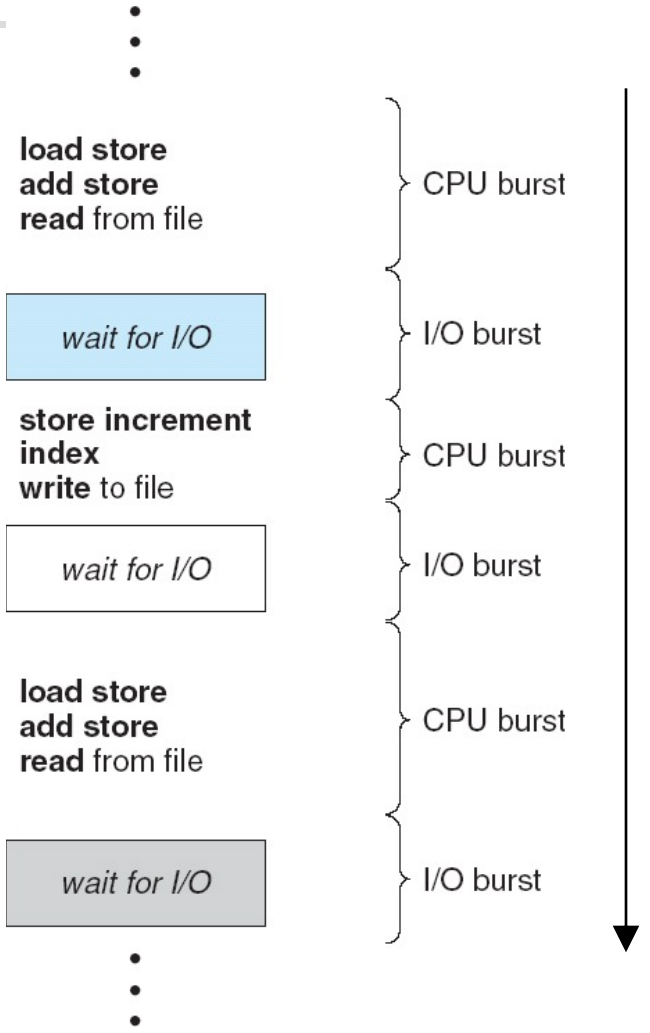
- Only one process/thread can run on a processor (or core) at a time.
  - All other processes have to wait until CPU (or a CPU core) is free and until they are scheduled to get CPU time.

- Two motivations for scheduling:
  - Multiprogramming (Batch) system: Maximise CPU utilisation and throughput of jobs.
    - While one process is blocked due to I/O, another process may use CPU.
  - Timesharing (Multitasking) system: Fast response time of all processes to allow users to work interactively.
    - While one process/thread is performing calculations, user can still interact with another process/thread because scheduler switches often between them.

Why Scheduling is Reasonable: CPU & IO Bursts

- Even though scheduling involves some overhead, it is effective to increase CPU utilisation:
  - CPU–I/O burst cycle: Process execution typically consists of a cycle of CPU usage and subsequent I/O wait (during which another process may use CPU).





Definitions: Scheduler, Scheduling algorithm, Dispatcher

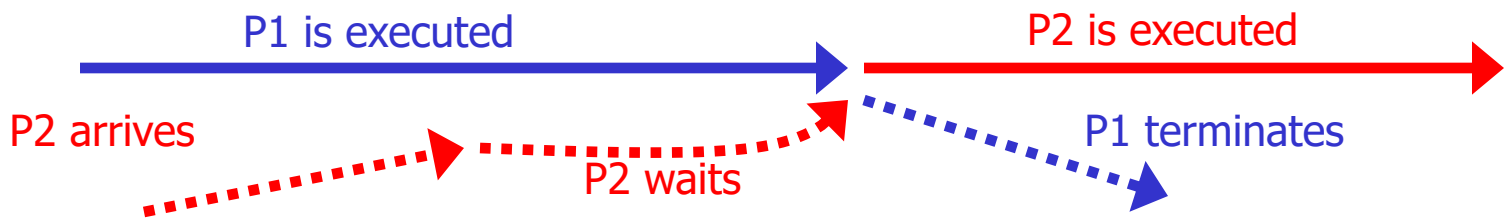
- (CPU) Scheduler:
  - Part of the OS kernel that assigns CPU time to processes/threads that are ready to execute.
- Dispatcher:
  - Part of the OS kernel that performs the actual context switch (restoring CPU registers of process, switching from kernel to user mode, resume execution of process)
    - Dispatch latency: time required for context switch (typically: 30µs).
- Scheduling algorithm:
  - Algorithm that is used by scheduler to decide which process/thread gets the CPU for how long.

Some Remarks

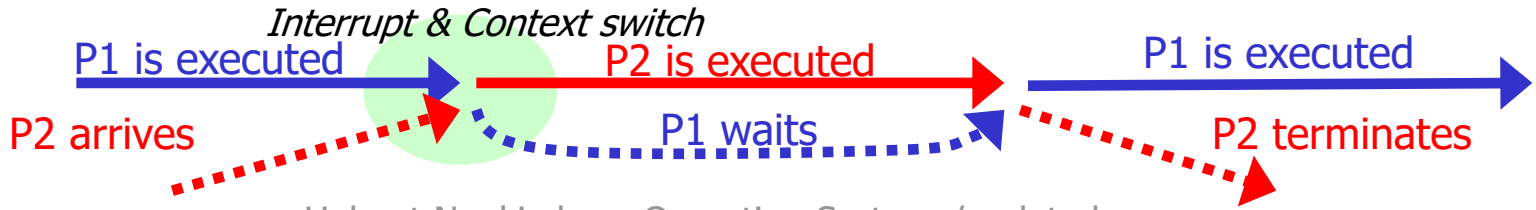
- Reminder from Section 2.6: Separate policy from mechanism!
  - Mechanism: Scheduler (& Dispatcher),
  - Policy: Scheduling algorithm.
- There is no "best" scheduling algorithm. It rather depends on the type of system (e.g. batch, multitasking, real-time) and scenarios.
- In Section 3.2 we also identified mid- and long-term schedulers: In this chapter, we will only consider short-term schedulers/CPU scheduling.
  - (Well, some can be used for long-term scheduling as well.)
- In operating systems with kernel-level threads, only threads (not processes) are scheduled. (Each process has at least one thread.)
  - In the following, we do not distinguish between processes and threads: "process scheduling" refers to threads as well.

Preemptive vs. Nonpreemptive Scheduling

- Nonpreemptive: CPU is allocated to process until it blocks or terminates.
  - Timesharing only possible if CPU-bound processes are cooperative and make calls to yield() system call (i.e. voluntary transition from running to ready state) to hand over control to scheduler. (Examples: MS Windows ≤3.x, Mac OS ≤9.x)



- Preemptive: Process may be interrupted, e.g. timesharing: time slice expired.
  - Problem: Process may be interrupted while updating data shared between processes: synchronisation between cooperating processes required to avoid inconsistencies.
  - Time slice timer may even expire while kernel code is executed: kernel must, e.g., disable interrupt processing while updating critical kernel data structures.



5.2 Scheduling Criteria (1)

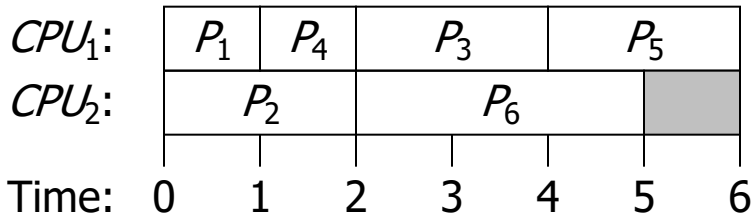
- Requirements on scheduling for all types of systems:
  - Fairness:
    - Each process gets CPU time (no starvation of processes).
  - Enforcement of priorities:
    - Processes with higher priorities are preferred.
  - Balance:
    - All the different resources of a system are reasonably utilised.
- Requirements on scheduling for multiprogramming (Batch) systems:
  - CPU utilization:
    - Maximise CPU utilization: keep the CPU as busy as possible.
  - Throughput:
    - Maximise number of processes that complete their execution per time unit.
  - Turnaround time:
    - Minimise amount of time (from start to termination) to execute a particular process.

Scheduling Criteria (2)

- Requirements on scheduling for timesharing (multitasking) systems, i.e. interactive systems:
  - Response time:
    - Minimise amount of time it takes from when a request was submitted until the first response is produced.
- Requirements on scheduling for real-time systems:
  - Meeting deadlines:
    - Meet time constraints: processes (or events within) that must be started/finished until a certain point in time must be preferred.
  - Predictability:
    - As long as the system is not overloaded (beyond an allowed level), it can be predicted, when a certain process (or event within) is executed.

Definitions (1)

- Schedule:
  - A schedule S for a set of cores/processors CPU = {CPU_1, CPU_2, ..., CPU_m} and a set of processes (tasks) P = {P_1, P_2, ..., P_n} (additionally, there might be dependencies between P_1, P_2, ..., P_m defined) with required service time d_1, d_2, ..., d_m is a mapping of processes to processors (or cores) with respect to time.
    - Note: in the following, we consider only service time that is provided by the CPU and do not talk about time spent while waiting due to blocking I/O.
- Visualisation of schedules for m = 2 CPUs (cores) using a Gantt chart:

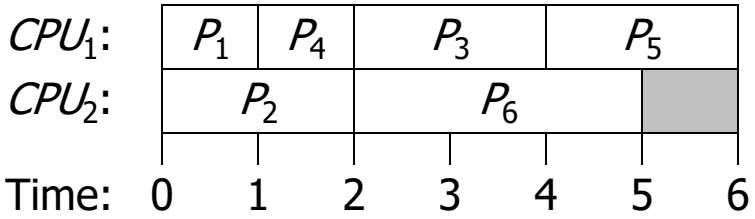


- The length l(S) of a schedule S is the time difference between the departure (finish time) of the final process and the arrival of the first process. (In the above example: l(S) = 6).

Definitions (2)

- Different points in time and durations for a process P_i:
  - a_i : Arrival time (when process is created/enters system)
  - d_i : Service time (duration of process if it would have CPU exclusively to do its calculations)
  - s_i : Start time (when process gets CPU for the first time)
  - f_i : Finish time (when process is completed/leaves system)
  - $r_i = f_i - a_i = w_i + d_i$: Residence time (Turnaround time) (time in system)
  - $w_i = r_i - d_i$: Waiting time (how long process is in system without being serviced)
  - $\tilde{W} = \frac{1}{n} * \sum_{i=1}^{n} w_i$: Average waiting time for n processes
  - $\tilde{R} = \frac{1}{n} * \sum_{i=1}^{n} r_i$: Average residence time for n processes

Application of Definitions: Examples

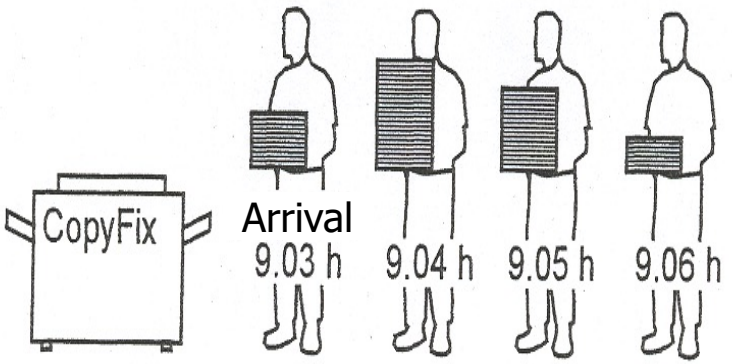


Process	Arrival time	Service time	Start time (= Waiting time, because nonpreemptive)	Residence time
P_1	$a_1 = 0$	$d_1 = 1$	$s_1 = w_1 = 0$	$r_1 = w_1 + d_1 = 1$
P_2	$a_2 = 0$	$d_2 = 2$	$s_2 = w_2 = 0$	$r_2 = w_2 + d_2 = 2$
P_3	$a_3 = 0$	$d_3 = 2$	$s_3 = w_3 = 2$	$r_3 = w_3 + d_3 = 4$
P_4	$a_4 = 0$	$d_4 = 1$	$s_4 = w_4 = 1$	$r_4 = w_4 + d_4 = 2$
P_5	$a_5 = 0$	$d_5 = 2$	$s_5 = w_5 = 4$	$r_5 = w_5 + d_5 = 6$
P_6	$a_6 = 0$	$d_6 = 3$	$s_6 = w_6 = 2$	$r_6 = w_6 + d_6 = 5$

- Average Waiting time: $\tilde{W} = \frac{1}{n} * \sum_{i=1}^{n} w_i = \frac{1}{6} * (0+0+2+1+4+2) = 1{,}5$
- Average Residence time: $\tilde{R} = \frac{1}{n} * \sum_{i=1}^{n} r_i = \frac{1}{6} * (1+2+4+2+6+5) = 3{,}\overline{3}$

5.3 Scheduling Algorithms: First Come First Served (FCFS)

- Principle:
  - Processes get CPU allocated in order of their arrival.
  - Running processes are not interrupted.
- Properties:
  - Nonpreemptive,
  - Fair (Finally, each process gets CPU),
  - Easy to implement.



- Remark:
  - Waiting times of each process depends on order of arrival.
  - Average waiting time may differ quite a lot when comparing best case and worst case arrival scenarios.
- General comment applying to all of the following algorithms: none of them waits until all processes have arrived, but immediately start with their first scheduling decision once a process arrived.

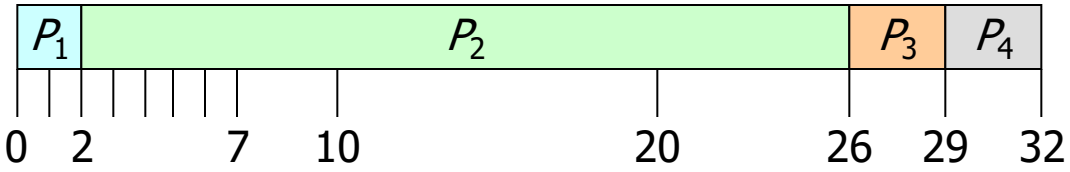
Scheduling Algorithms: First Come First Served (FCFS)

- Example scenario used on next slides (also used to illustrate the further scheduling algorithms):
  - 4 processes,
  - 1 CPU (core),
  - No dependencies between processes.

Process	Arrival time	Service time
P_1	$a_1 = 0$	$d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$
P_3	$a_3 = 2$	$d_3 = 3$
P_4	$a_4 = 7$	$d_4 = 3$

Scheduling Algorithms: First Come First Served (FCFS)

- Schedule with FCFS:



- As table:

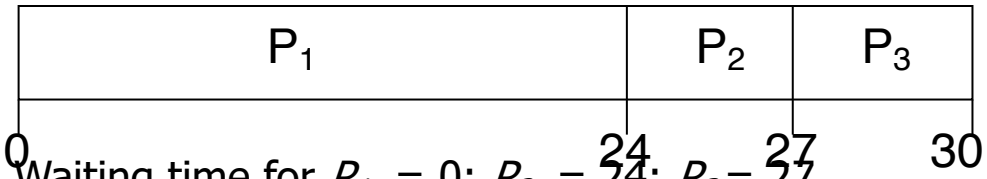
Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 2$	$s_1 = 0$	$w_1 = s_1 - a_1 = 0$	$r_1 = w_1 + d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$	$s_2 = 2$	$w_2 = s_2 - a_2 = 1$	$r_2 = w_2 + d_2 = 25$
P_3	$a_3 = 2$	$d_3 = 3$	$s_3 = 26$	$w_3 = s_3 - a_3 = 24$	$r_3 = w_3 + d_3 = 27$
P_4	$a_4 = 7$	$d_4 = 3$	$s_4 = 29$	$w_4 = s_4 - a_4 = 22$	$r_4 = w_4 + d_4 = 25$

- Average waiting time $\tilde{W} = \frac{1}{n} * \sum_{i=1}^{n} w_i = \frac{1}{4} * (0 + 1 + 24 + 22) = 11{,}75$

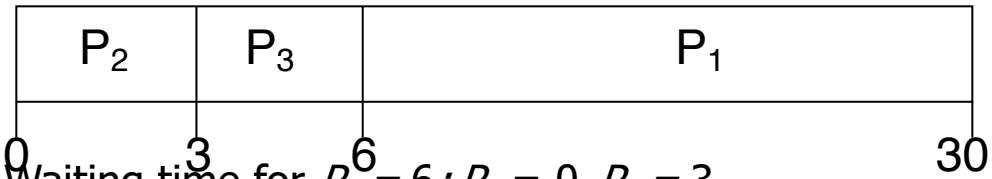
- Average residence time $\tilde{R} = \frac{1}{n} * \sum_{i=1}^{n} r_i = \frac{1}{4} * (2 + 25 + 27 + 25) = 19{,}75$

Scheduling Algorithms: First Come First Served (FCFS)

- Waiting times (and thus also residence times) depend very much on order of process arrival!
- Example: d_1=24, d_2=3, d_3=3, two different orders of process arrival:
  - Processes arrive at a_i=0 in the order: P_1 , P_2 , P_3



    - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
    - Average waiting time: (0 + 24 + 27)/3 = 17
  - Processes arrive at a_i=0 in the order: P_2 , P_3 , P_1

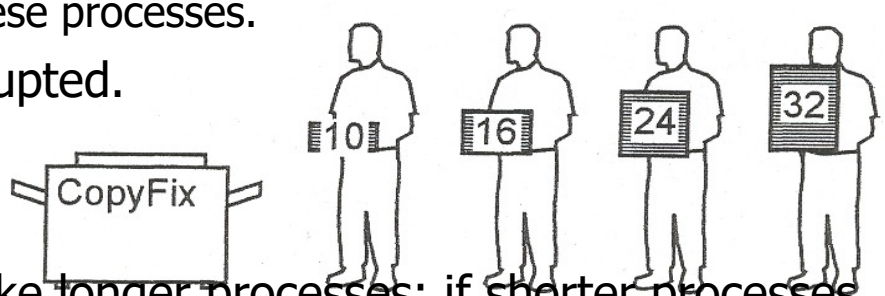


    - Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
    - Average waiting time: (6 + 0 + 3)/3 = 3

This insight leads us to the scheduling algorithm on the next slide…

Scheduling Algorithms: Shortest Job First (SJF)

- Principle:
  - Considers service time of process: process with shortest service time gets CPU first.
    - Same service times: FCFS for these processes.
  - Running processes are not interrupted.
- Properties:
  - Nonpreemptive.
  - Unfair (short process may overtake longer processes: if shorter processes keep are arriving all the time, longer process will suffer from starvation, i.e. will never get serviced.)
- Remark:
  - Optimises (provable) average waiting time of processes (nonpreemptive).
  - However, problems remain:
    - Unfair,
    - Service time must be known in advance.

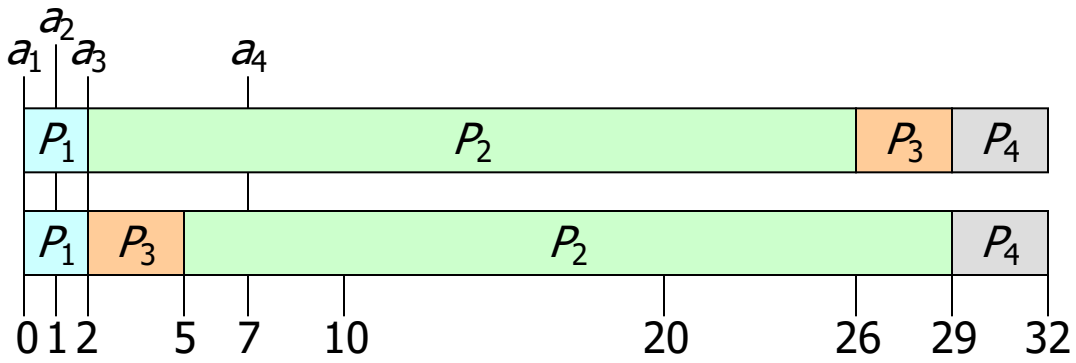


Excursion: How to Know Service Time of a Process in Advance?

- Typically, service time of a process is not known in advance.
  - At least, a scheduler cannot predict.
- However, in batch systems (e.g. supercomputers running jobs over night), human submitters of batch jobs often need to specify a time limit for their job.
  - SJF can be used (and is often used) for long-term scheduling (job scheduling) in batch systems.
  - User's motivation for estimating time limit as good as possible:
    - If specified time limit is too low, job might exceed time limit and gets therefore terminated (to detect hanging jobs).
    - If specified time limit is too high, job will have to wait long until SJF will give it CPU time.
- Later on, we will see an example how to predict the length of CPU bursts to use this for SJF-like short-term scheduling.

Scheduling Algorithms: Shortest Job First (SJF)

- For comparison: Schedule with FCFS:
- Schedule with SJF:



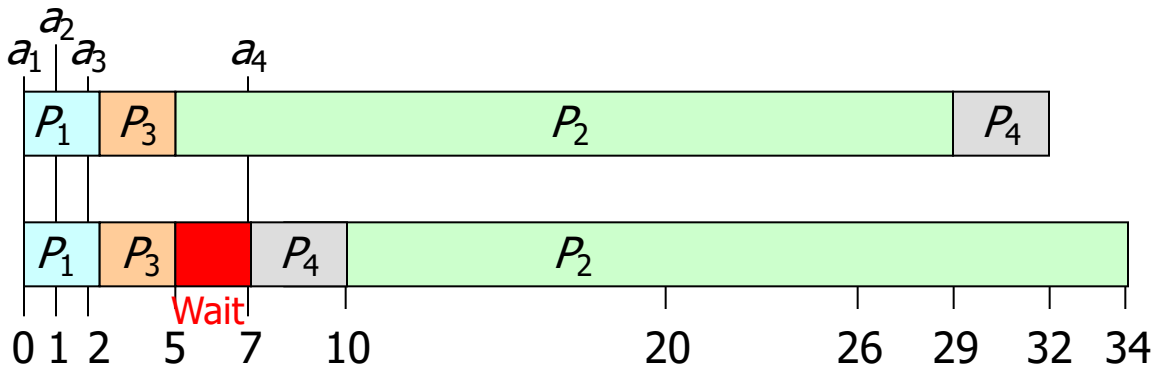
- Table:

Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 2$	$s_1 = 0$	$w_1 = s_1 - a_1 = 0$	$r_1 = w_1 + d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$	$s_2 = 5$	$w_2 = s_2 - a_2 = 4$	$r_2 = w_2 + d_2 = 28$
P_3	$a_3 = 2$	$d_3 = 3$	$s_3 = 2$	$w_3 = s_3 - a_3 = 0$	$r_3 = w_3 + d_3 = 3$
P_4	$a_4 = 7$	$d_4 = 3$	$s_4 = 29$	$w_4 = s_4 - a_4 = 22$	$r_4 = w_4 + d_4 = 25$

- Average waiting time $\tilde{W} = \frac{1}{n} * \sum_{i=1}^{n} w_i = \frac{1}{4} * (0+4+0+22) = 6{,}5$
- Average residence time $\tilde{R} = \frac{1}{n} * \sum_{i=1}^{n} r_i = \frac{1}{4} * (2+28+3+25) = 14{,}5$

Scheduling Algorithms: Shortest Job First (SJF) with Waiting

- Paradox, if SJF decides to do nothing (waiting):
- Schedule with SJF:
- Schedule with SJF & waiting:
- Table:



Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 2$	$s_1 = 0$	$w_1 = s_1 - a_1 = 0$	$r_1 = w_1 + d_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$	$s_2 = 10$	$w_2 = s_2 - a_2 = 9$	$r_2 = w_2 + d_2 = 33$
P_3	$a_3 = 2$	$d_3 = 3$	$s_3 = 2$	$w_3 = s_3 - a_3 = 0$	$r_3 = w_3 + d_3 = 3$
P_4	$a_4 = 7$	$d_4 = 3$	$s_4 = 7$	$w_4 = s_4 - a_4 = 0$	$r_4 = w_4 + d_4 = 3$

- Average waiting time $\tilde{W} = \frac{1}{n} * \sum_{i=1}^{n} w_i = \frac{1}{4} * (0+9+0+0) = 2{,}25$
- Average residence time $\tilde{R} = \frac{1}{n} * \sum_{i=1}^{n} r_i = \frac{1}{4} * (2+33+3+3) = 10{,}25$

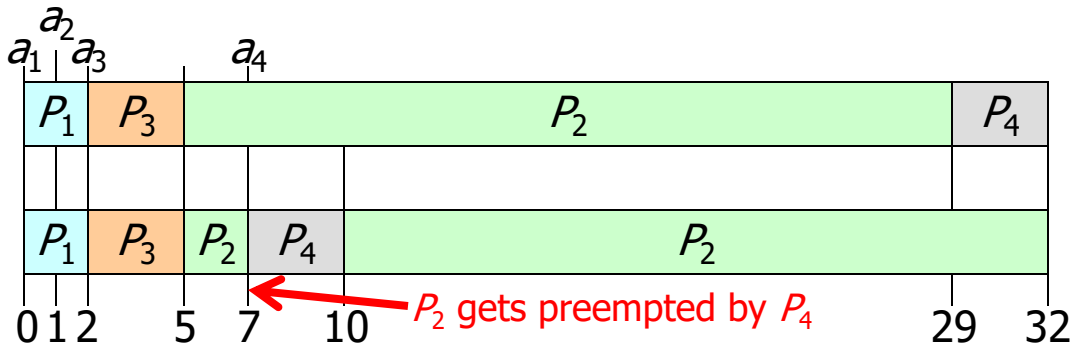
Got shorter! (However, requires global knowledge to know future.)

Scheduling Algorithms: Shortest Remaining Time First (SRTF)

- Principle:
  - At any time, the process with the shortest remaining time gets the CPU.
    - Preemptive variant of SJF (based on insight from the paradox on previous slide): When a new process arrives and service time of that process is shorter than the remaining service time of the running process, the running process will be preempted and the just arrived process gets the CPU.
  - Running processes may be interrupted.
- Properties:
  - Preemptive.
  - Unfair (short process may overtake longer processes: if shorter processes keep are arriving all the time, longer process will suffer from starvation.)
- Remark:
  - Optimises (provable) average waiting time of processes (preemptive).
  - However, problems remain:
    - Unfair,
    - Service time must be known in advance.

Scheduling Algorithms: Shortest Remaining Time First (SRTF)

- For comparison: Schedule with SJF (without waiting):
- Schedule with SRTF
- Table:



Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 2$	$s_1 = 0$	$w_1 = 0$	$r_1 = 2$
P_2	$a_2 = 1$	$d_2 = 24$	$s_2 = 5$	$w_2 = 4 + \mathbf{3} = 7$ (preempted by P_4)	$r_2 = 31$
P_3	$a_3 = 2$	$d_3 = 3$	$s_3 = 2$	$w_3 = 0$	$r_3 = 3$
P_4	$a_4 = 7$	$d_4 = 3$	$s_4 = 7$	$w_4 = 0$	$r_4 = 3$

- Average waiting time $\tilde{W} = \frac{1}{n} * \sum_{i=1}^{n} w_i = \frac{1}{4} * (0+7+0+0) = 1{,}75$
- Average residence time $\tilde{R} = \frac{1}{n} * \sum_{i=1}^{n} r_i = \frac{1}{4} * (2+31+3+3) = 9{,}75$

Scheduling Algorithms: Interactive Timesharing Systems

- The previously discussed scheduling algorithms are mainly applicable for batch systems:
  - Processes run from start to completion and are typically not interrupted (except for I/O or – in case of SRTF – when a new process arrives.)

- To support interactive systems that give the user the impression that multiple processes are running in parallel, preemptive scheduling algorithms are required:
  - CPU time is divided into slices that are periodically allocated to processes based on criteria that are specific to each scheduling algorithm (→following slides).
    - A keypress will be delivered after a couple of time slices to a process waiting for it even if all other processes are only using the CPU.

Scheduling Algorithms: Round-Robin (RR) Scheduling

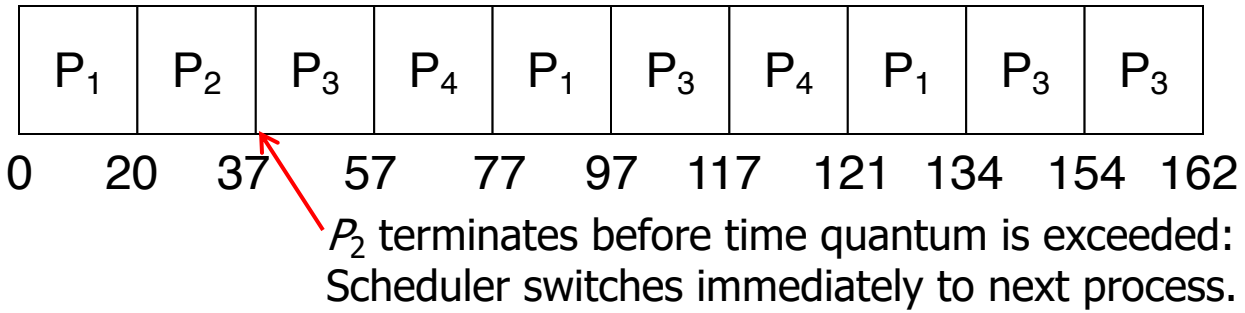
- Principle:
  - Service time is divided into time slices (with a maximum duration of a time quantum – actual slice may be shorter, e.g. if process does I/O →5-5).
  - Ready queue is a circular FIFO buffer (like FCFS, but circular): scheduler assigns time slice to process at head of ready queue.
    - Either: process makes blocking I/O before time slice expires: process is put at the end of the waiting queue.
    - Or: CPU burst of process is longer than time slice (timer interrupt preempts process): process is put at the end of the ready queue.
    - Newly arriving processes are put at the end of the ready queue.
- Properties:
  - Preemptive.
  - CPU time is equally distributed to all processes (fair).

Otherwise, old process would suffer from starvation if constantly new processes arrive.

- Remark:
  - Variations possible with respect to time quantum:
    - Long time quantum (>100ms): slow interactions. (E.g.: 10 processes or users, and 100ms time slices: User has to wait 1s for next interaction with process.)
    - Short time quantum (<10ms): large overhead due to frequent context switches.

Scheduling Algorithms: Round-Robin (RR) Scheduling

- Example (different scenario than before):
  - Service times: d_1=53, d_2=17, d_3=68, d_4=24
  - Processes arrive all at t=0 in the order: P_1 , P_2 , P_3, P_4
  - Schedule with RR and a time quantum of 20:

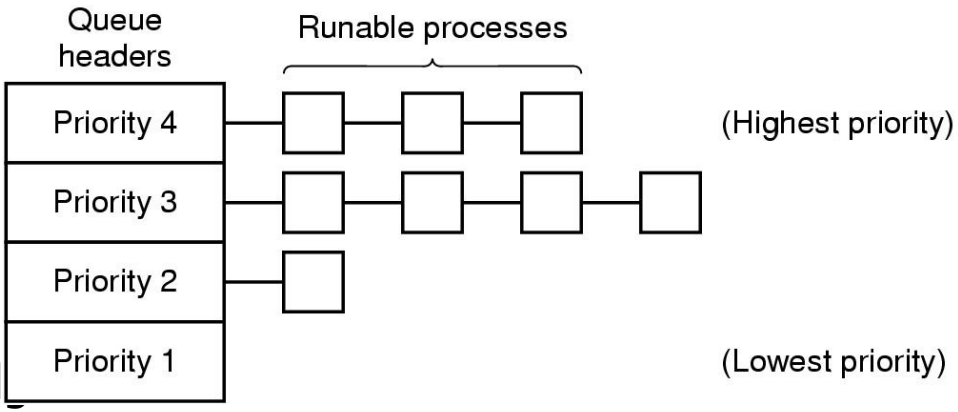


As most CPU-bursts are <20ms (→5-5), it is typically the case with 20ms time quantum that the used times slice is shorter than the time quantum.

Process	Arrival time	Service time	Start time	Waiting time	Residence time
P_1	$a_1 = 0$	$d_1 = 53$	$s_1 = 0$	$w_1 = 0+57+24 = 81$	$r_1 = w_1 + d_1 = 134$
P_2	$a_2 = 0$	$d_2 = 17$	$s_2 = 20$	$w_2 = 20$	$r_2 = w_2 + d_2 = 37$
P_3	$a_3 = 0$	$d_3 = 68$	$s_3 = 37$	$w_3 = 37+40+17 = 94$	$r_3 = w_3 + d_3 = 162$
P_4	$a_4 = 0$	$d_4 = 24$	$s_4 = 57$	$w_4 = 57+40 = 97$	$r_4 = w_4 + d_4 = 121$

Scheduling Algorithms: Round-Robin with Priorities

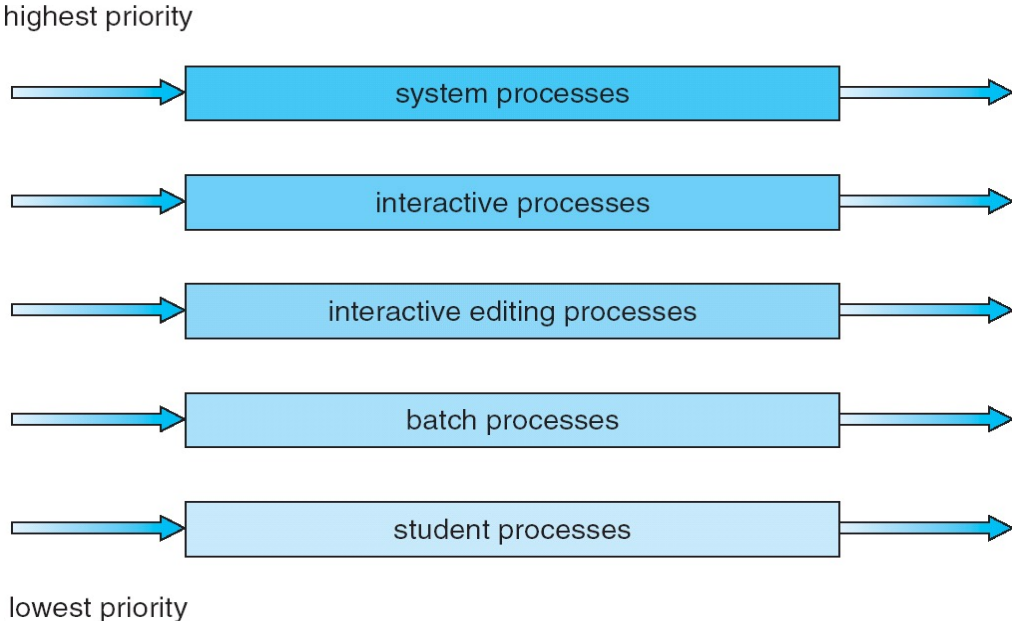
- Principle:
  - Processes have priorities and each priority has its own ready queue.
  - First, all processes from ready queue with highest priority are scheduled using Round Robin al
  - Only if queue is empty (because all processes terminated or are blocked), the ready queue with the next lower priority is served and so on.
    - However, even if a higher priority process arrives, a currently running lower priority process gets not preempted, until its time slice expires anyway.



- Properties:
  - Low-priority processes may suffer from starvation: if new high priority processes are arriving steadily (or do never block or terminate), then low priority processes will never get executed.
  - Solution: Aging: priorities of processes are dynamically adjusted:
    - Priority decreases with increasing residence time.
    - Priority increases with increasing waiting time.

Scheduling Algorithms: Multilevel Queue Scheduling

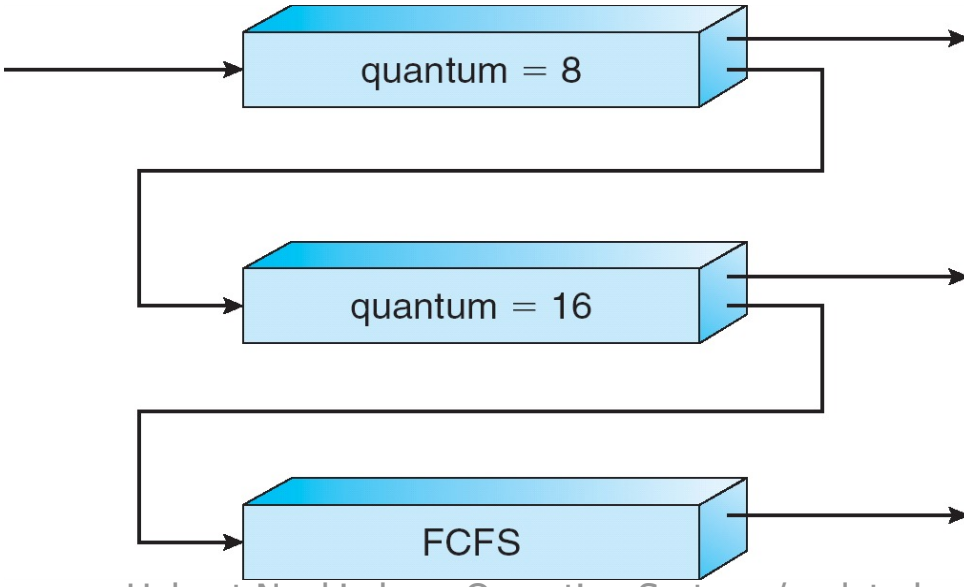
- Principle:
  - Different queues have different scheduling algorithms, e.g.
    - queues for interactive foreground processes using RR scheduler.
    - queues for batch background processes using FCFS scheduler.



  - Scheduler between queues required: e.g. foreground queue has absolute priority: no background process will run unless other queues are empty.

Scheduling Algorithms:
Multilevel Feedback Queue Scheduling

- Principle:
  - Like Multilevel Queue Scheduling, however processes can move between the various queues.
    - Aging can be implemented this way:
    - if a process uses too much CPU time, it will move to a lower-priority queue.
    - if a process waits for long time, it will move to a higher-priority queue.



5.5 Thread Scheduling

- If only user-level threads are used, the OS kernel is not aware of these threads, but simply schedules the processes.
  - Scheduling of user-level threads takes place within user-level thread library.
  - Threads of the same process are competing for CPU time that has been allocated by the OS scheduler to that process: process-contention scope (PCS).

- However, if kernel-level threads are used, the OS kernel basically schedules threads, not processes.
  - Typically, OS scheduler does not care to which process all the threads belong. Instead, all threads of the system compete equally for CPU time: system-contention scope (SCS).
  - Most OS that use the one-to-one multithreading model, use SCS.

5.6 Multiple-Processor Scheduling/ Multi-core Scheduling

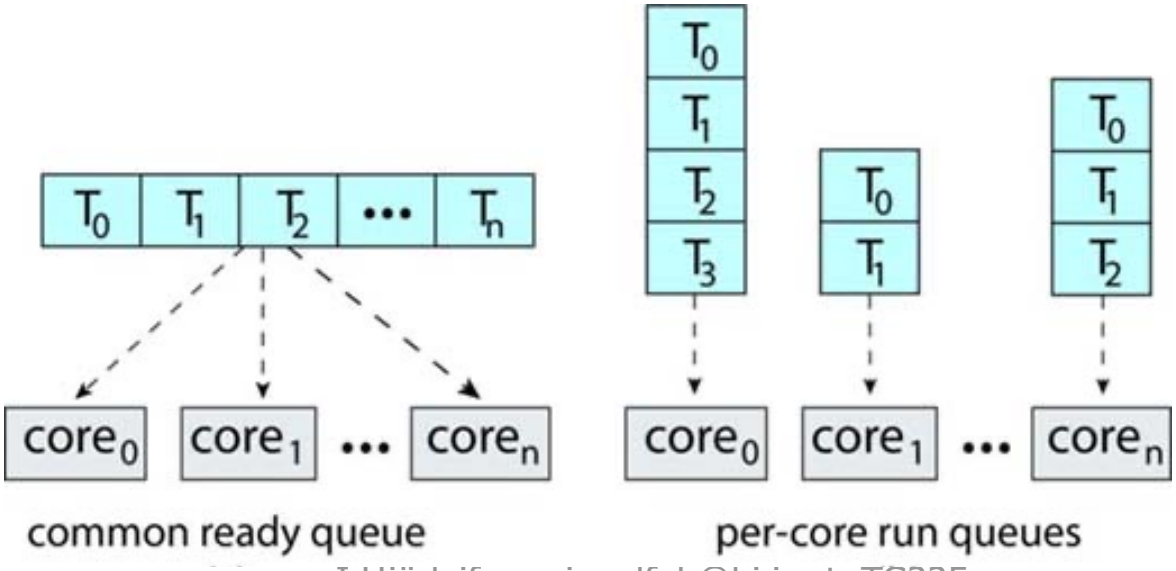
- CPU scheduling more complex when multiple CPUs/cores that share the physical memory are available.
  - Each CPU/core must be managed by the operating system.

- Asymmetric multiprocessing: only one master processor/core accesses the kernel data structures (e.g. scheduler queues) and makes scheduling decisions, the others are waiting to get work assigned.
  - Master processor runs fully-fledged kernel,
  - Slave processors only minimal kernel.

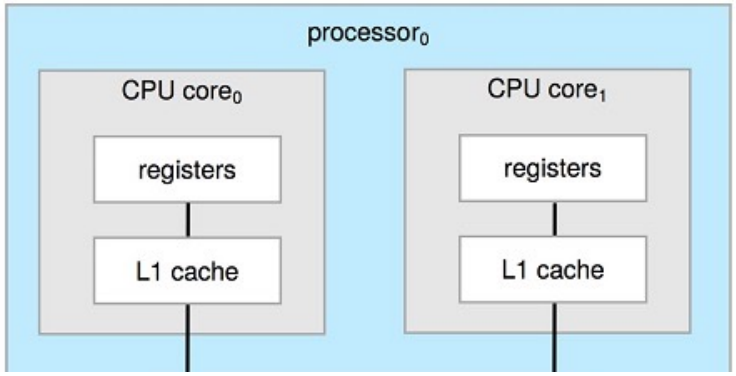
- Symmetric multiprocessing (SMP): all processors/cores are running the same kernel. I.e. each processor/core executes scheduler instructions and selects ready processes to be executed.
  - Nowadays, SMP used by all major operating systems.

Symmetric Multiprocessing and Scheduler Queues

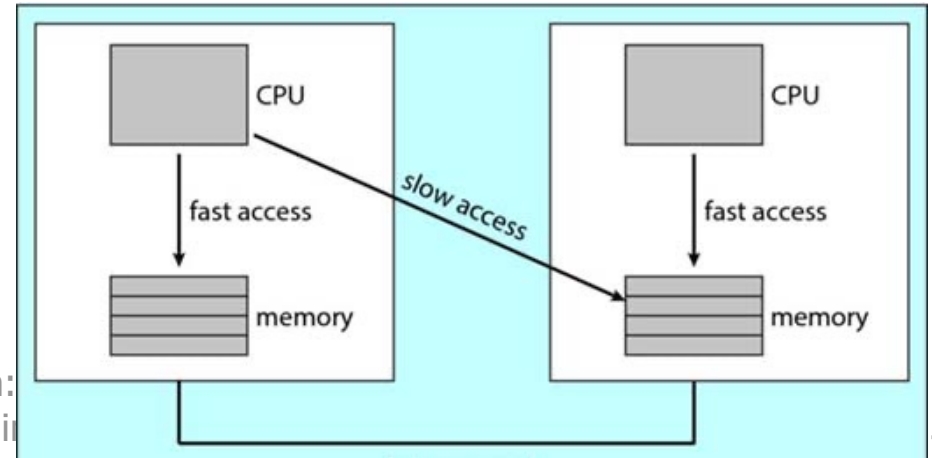
- Two ways of organising scheduler queues in SMP:
  - One common ready queue for all processors/cores:
    - A process from the ready queue may be run by any of the available CPUs/cores.
    - All processors/cores maintain concurrently the shared kernel data structures.
      - Synchronisation (→ch. 6) required to avoid inconsistencies of kernel data structures.
  - Each processor/core has its own ready queue:
    - Queues are "private" to each CPU/core.
    - No synchronisation needed for access to queues needed.



Processor Affinity



- Each CPU/core has its own cache containing recently used instructions/data.
  - If the scheduler assigns a process to a CPU/core that has previously executed a different process, that chosen CPU/core will not have the instructions and data of the newly assigned process in its cache.
  - ⇒ No benefit (=no speedup) from caching if a process is migrated to a different processor (core) at each scheduling decision.
  - Processor affinity: scheduler tries to keep a process on the same processor/core.
- Even worse in Non-Uniform Memory Access (NUMA) multiprocessor systems:
  - Each CPU of a multiprocessor computer has it's own RAM: fast access.
  - Still a CPU can access the RAM of the other CPU, but slower.
  - NUMA used in supercomputing:
    - One motherboard contains two multicore CPUs with fast access to it's RAM.
    - Shared memory communication possible by accessing RAM of other CPU (=slower than own RAM, but still faster than communication via network).

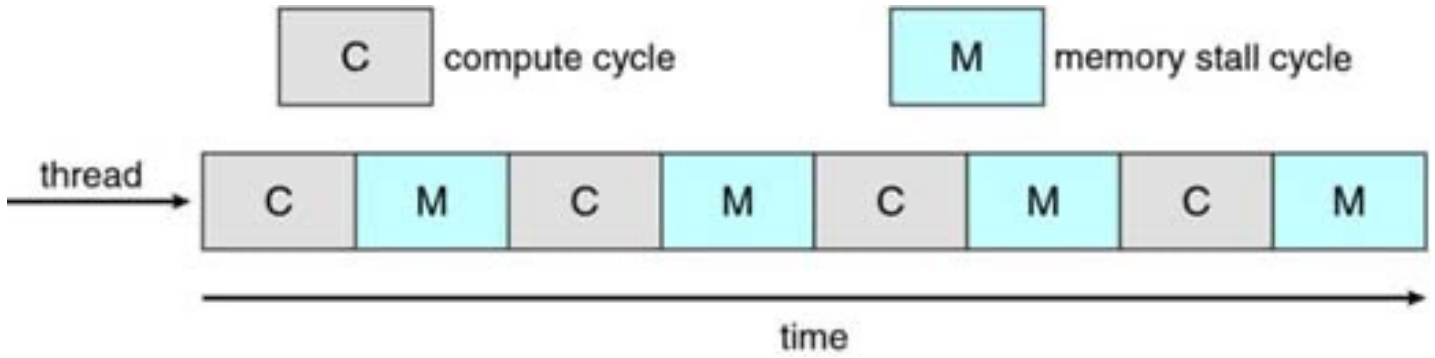


Processor Affinity vs. Load Balancing

- Hard processor affinity:
  - Process migrates never between CPUs/cores
  - E.g. when each CPU/core has its own, "private" ready queue.
  - Disadvantage: some CPUs/cores may be busy executing processes while others are idle even though ready processes are waiting in a queue.
- Load balancing attempts to distribute the workload between CPUs/cores.
  - Push migration checks periodically load on each processor/core and migrates ("pushes") processes if necessary.
  - Pull migration is performed whenever a processor/core is idle (=queue becomes empty). Then, the idle processor/core pulls a process from another processor's/core's queue.
- Processor affinity and load balancing contradict each other.
  - Difficult to develop scheduling algorithms that achieve a good compromise.
  - Soft processor affinity: try to maintain affinity, but allow load balancing.

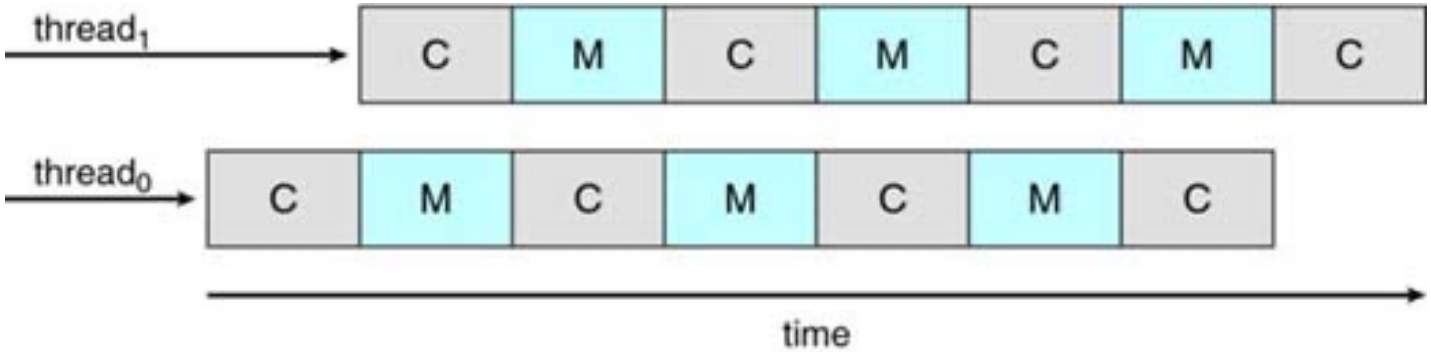
Memory stalls

- On OS-level, processes get blocked if they are waiting for I/O – Same may happen on CPU level:
  - Accessing main memory is slow (in comparison to a CPU cycle).
  - While a machine code instruction is waiting for RAM access ("memory stall"), the CPU is idle.
    - In practise, a CPU may be waiting approx. 50% of the time for RAM access.
  - Countermeasures:
    - Bigger caches inside CPU (=faster than RAM),
    - "Hyperthreading" →next slide.



"Hyper-Threading"/Hardware multithreading/ Simultaneous Multi Threading (SMT)/Chip multithreading (CMT)

- Just like OS-level, when a process is blocked because waiting for I/O and OS scheduler selects another ready process to execute instead
  – Do the same on CPU level:
- "Hyper-Threading" (Intel marketing term), also known as Hardware multithreading, Simultaneous Multi Threading (SMT), or Chip multithreading:
  - A CPU core gives the impression that it is actually two (or even more) CPU cores.
  - In fact, it is only one physical core that is just able to switch to another thread of instructions in case of a memory stall, so that one physical core actually executes two threads by pretending to be two logical cores.
  - For switching, CPU contains some sort of "scheduler" in the CPU hardware, e.g. save state (e.g. CPU registers) of hardware thread inside CPU hardware.



“Hyper-Threading” and Scheduling on Multicore Processors

- Problem: Even the OS may not be aware of hyper-threading and treats logical cores just like real physical cores.
  - Problem, e.g. dual core processor providing four logical cores, but only two processes are currently running:
    - OS CPU scheduler might accidentally choose to schedule these two processes on those two logical cores that belong to the same physical core.
    - That physical core would become extremely busy while the other physical core would be idle, leading to slower speed than is possible with better scheduling.

  ⇒ Make OS scheduler aware of hyper-threading!
    - So that the OS scheduler can do load balancing based on physical cores, not logical cores.

5.7 Operating System Examples

- In Unix-like systems (e.g. Linux), the scheduling algorithms are subject of frequent changes: often, each new kernel version has a new scheduler (However, typically just variants of RR with priorities).
  - ⇒ We do not cover Linux scheduling here as it is a moving target. (If you have a printed book that covers the Linux scheduler, you can be sure that it is outdated when you buy the book…)

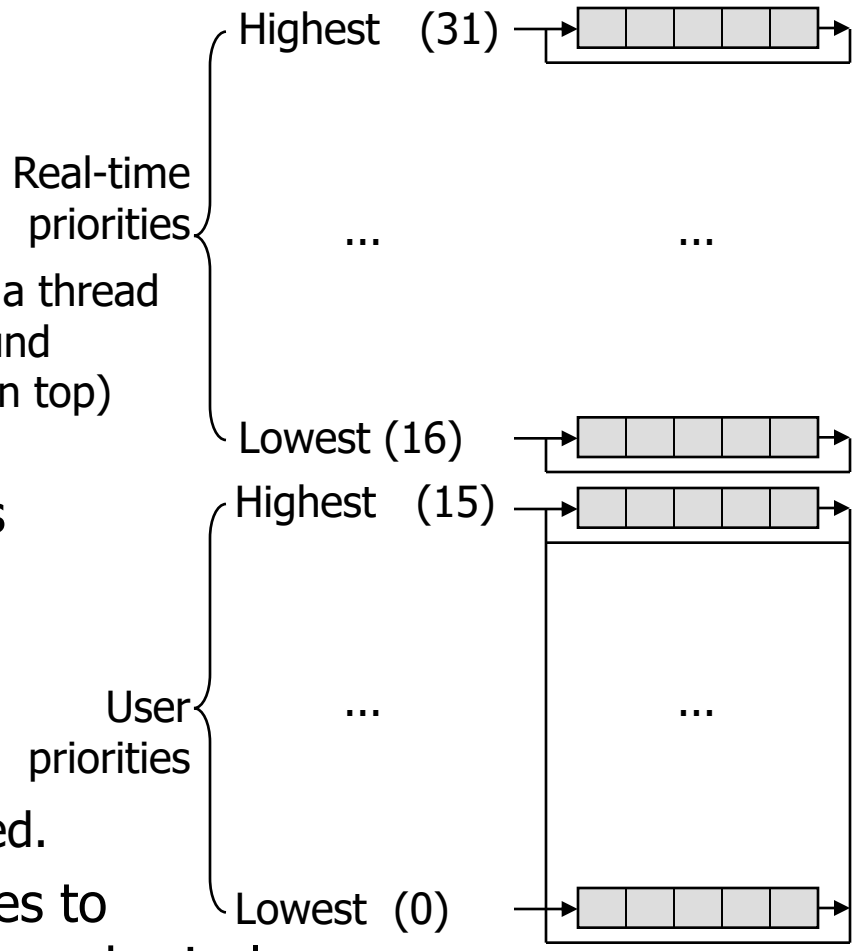
- Scheduling algorithms of Microsoft Windows do not change that frequently.
  - ⇒ We will have a closer look on the Microsoft Windows scheduling algorithm (used since Windows XP and later) on the next slides.
  - Since Windows 7, user-level-like threads are supported natively (user-mode-scheduling, UMS) in the kernel and for them, own schedulers can be provided.

Microsoft Windows Scheduling

- Kernel-level threads, i.e. threads are scheduled with system-contention scope.
- Preemptive Multilevel Feedback Queue scheduling based on 32 different priorities:
  - 16 real time (or system) priorities (16-31):
    - For operating system threads or for threads to which the administrator assigned real-time priority.
    - Priorities in this class do not change.
  - 16 user priorities (0-15, 0 only used for low priority system threads):
    - Priorities may be set by a user (UI) or a process/thread (API).
    - Priorities in this class change over the time.
  - Soft-real time: If a higher priority real-time thread becomes ready (e.g. I/O completed) while another thread is running, that other is immediately preempted. (However, no guaranteed deadlines.)

Microsoft Windows Scheduler ("Dispatcher")

- Round-Robin with priorities
  - Time quantum:
    - Windows Server: 120ms
    - Desktop editions: 20ms
      - Typically, the time quantum of a thread that is running in the fore-ground (i.e. window of application is on top) is multiplied by a factor of 3.
- Scheduler manages 32 FIFO queues containing ready threads of the respective priority.
  - Threads with lower priorities only get CPU allocated when all higher priority threads are blocked.
- Processor affinity: The scheduler tries to schedule a thread always on the same physical core.



Dynamic Priorities for Microsoft Windows User Processes

- Priorities of non real-time threads are dynamically adjusted:
  - Priority is increased (however, never >15 which would be real-time) when a blocked thread gets ready again.
    - ⇒Utilisation of I/O devices increases (a thread that had to wait for I/O in the past, is likely to do I/O in the future again).
    - ⇒Response time of interactive thread decreases (=gets faster).
      - Threads that have been blocked because of user interaction (waiting for keyboard, mouse) get a larger increase in priority than other I/O.
      - In addition, threads belonging to a window that is in the foreground get a priority boost (in addition to the boost of the time quantum).
    - After each time slice, priority of the expired thread decreases by one unti initial priority is reached.
  - A thread that did not run for some amount of time (i.e. that might suffer from starvation), gets its priority set to 15 for 2 time slices.

5.9 Summary

- CPU scheduling: select a waiting process from the ready queue and allocate the CPU to it.
- First-come, first-served (FCFS): simple.
- Shortest-job-first (SJF): optimises average waiting time, but starvation possible (unfair).
  - However, length of job often not known in advance.
    (Shortest Burst Next: Tries to predict length of next burst using exponential averaging.)
- Shortest-remaining-time-first (SRTF): preemptive variant of SJF.
- Round-Robin (RR): appropriate for interactive systems due to preemptive time slicing.
- Round-Robin with priorities: supports priorities, however starvation may occur (unfair).
  - Aging of process priorities may prevent starvation.
- Multilevel Queue algorithms use different algorithms for different classes of processes.
  - Feedback variant allows to move processes between queues.
- In practise, typically Round-Robin with priorities or Multilevel Feedback Queue Scheduling (often involving Round-Robin) is used (see e.g. Windows XP scheduling).
- Symmetric multiprocessing (SMP): each processor executes the same instructions of the kernel and access the same shared kernel data structures, e.g. scheduling queues.
- Advanced topic not treated in this course: Evaluation of scheduling algorithms
  - Analysis based on queueing-network theory, simulation, or implementation.