# Process Scheduling

## Agenda

- Recap
- General scheduling mechanism
- Metrics and goals of scheduling
- Scheduling algorithms
- Multiprocessor scheduling
- Case studies: UNIX, Linux, Solaris, Windows XP
- Scheduling in thread libraries
- Evaluating scheduling algorithms

## Recap

- Concept of process
  - Loosely speaking, anything that runs on the CPU
- System calls to control what a process does (other than computing)
  - Creation of new process/thread
  - Communicating with another process/thread: message passing, shared memory, event signal
  - Saving and restoring the context of a process
- Multithreading libraries

## General scheduling mechanism

- Process scheduling is the activity of selecting the process that will run next on the CPU
- If the scheduler needs to run in the kernel mode, there has to be a mode switch in the currently running process before scheduling can take place
  - The mode switch is usually a result of a system call or an interrupt
  - A scheduling decision may have to be taken only on a long-latency system call and some interrupts such as the timer interrupt
- A scheduler saves the context of the currently running process, selects a process from the ready queue, restores the context of selected process

## General scheduling mechanism

- The scheduler can be invoked in four possible circumstances
  - The currently running process goes on a long-latency system call i.e. transitions from running to sleep state
  - A new process is created or a process completes a long-latency system call i.e., a process transitions from created to ready or from sleep to ready
  - The currently running process terminates
  - The currently running process receives a timer interrupt
  - The first and the third cases lead to non-preemptive or co-operative scheduling
  - The remaining two cases lead to pre-emptive scheduling

## Goals of process scheduling

- A scheduling algorithm can target a subset of the following
  - Maximize throughput: rate at which processes complete
  - Minimize turnaround time
    - Average, maximum, standard deviation?
  - Minimize waiting time in the ready queue
    - Direct measure of scheduler efficiency
    - Average, maximum, standard deviation?
  - Minimize response time
    - How long a process takes to produce the first result
    - Important for interactive systems
    - Average, maximum, standard deviation?

## Scheduling algorithms: FCFS

- Non-preemptive first-come-first-serve
- May lead to a convoy effect if one process has large CPU bursts and others have small CPU bursts
- In the worst case, FCFS has an unbounded average waiting time

## Scheduling algorithm: SJF

- Non-preemptive shortest next CPU burst scheduling
  - Popularly known as shortest job first scheduling
  - Provably optimal for average waiting time and average turnaround time
  - Drawback: requires knowledge about future CPU bursts
  - One popular way of estimating CPU bursts is exponential averaging: $s(n+1) = at(n) + (1-a)s(n)$
    - Need to start with a guess for $s(0)$, but little effect in long run
  - Pre-emptive version is called shortest remaining time first (SRTF)

## Scheduling algorithms: Priority

- Each process is assigned a priority
  - Either by kernel based on the resource usage profile of the process or externally by the user
- The process with the highest priority is scheduled
  - Can be pre-emptive or non-preemptive
- A steady flow of CPU bursts from the high-priority processes can starve the low-priority ones
  - Age-based priority modulation solves this problem
- SJF is a special-case priority scheduling policy

## Scheduling algorithms: Round-robin

- Pre-emptive quantum scheduling
  - The ready queue is treated as a circular FIFO and each process in the FIFO order is assigned a fixed time slice or quantum for execution
  - If the running process goes to sleep or terminates before the expiry of the quantum, the next process in the FIFO order is scheduled
  - If the running process's quantum expires, it is put back at the tail of the ready queue
  - The time quantum should be chosen to be larger than the context switch overhead
    - Too large a time quantum leads to FCFS sheduling
    - 80% of the CPU bursts should be shorter than the time quantum: is this thumb rule useful?

## Scheduling algorithms: Multi-level queues

- Often it is necessary to design different scheduling policies for different types of processes
  - Foreground processes would require a scheduling policy that honours time-sharing e.g., round-robin
  - Background processes are happy with something as simple as FCFS
- One possible implementation: maintain multiple ready queues each having its own policy
  - One for each type of processes
  - Order queues by priority for scheduling across queues

## Scheduling algorithms: Multi-level queues

- Multi-level queue scheduling
  - Round-robin scheduling across queues
  - Quantum attached to a queue is proportional to its priority
    - Processes from higher-priority queues get to run longer
  - Within a queue, the scheduling algorithm depends on the type of the processes in the queue
  - Drawback: a process cannot migrate from one queue to another even if its behavior has changed
    - The priority of a process may change over its life time
    - Solution: multi-level feedback queues

## Scheduling algorithms: Multi-level queues

- Multi-level feedback queue scheduling: an example
  - Suppose we have three queues with three different scheduling quanta
  - A new process is always enqueued at the tail of the queue with the smallest quantum
  - If a process fails to complete its current CPU burst within the time quantum allocated for its current queue, the process is enqueued at the tail of the queue with the next higher time quantum
    - The queue with the largest time quantum could just execute non-preemptive FCFS
  - The frequency of scheduling a queue must be inversely proportional to the time quantum

## Scheduling algorithms: Deadlines

- In real-time systems, usually a deadline is attached with each process
  - Scheduling algorithm needs to minimize the average, maximum, or standard deviation of overshoot
  - Two types of processes
    - With hard deadlines (must be met)
    - With soft deadlines (minimize some function of overshoot)
  - Tempting to sort the processes by deadline and scheduling them in earliest-deadline-first order
    - No guarantee on overshoot if all the deadlines cannot be met
  - How to handle a continuous flow of processes?
    - All deadlines not known a priori (pre-emptive EDF)

## Scheduling algorithms: Deadlines

- How to make sure that the owner of a process submits the true deadline?
  - Possible to submit an earlier deadline than the actual to gain priority in scheduling
    - Particularly problematic if the user knows that the scheduling algorithm is EDF
    - Any priority scheme that depends on deadline alone will suffer
- No solution would be perfect in this case
  - If there are too many processes, the scheduler could switch to round-robin
  - Important for the user to specify correct deadlines when the hard deadlines are mission-critical

## Multiprocessor scheduling

- Two important aspects: load balancing and data affinity
- Where does the OS code run? Options:
  - A group of processors dedicated to carry out kernel activities (known as asymmetric scheduling)
  - A single OS node simplifies OS design and improves performance
  - The OS code runs on the processor that asks for a kernel service (known as symmetric scheduling)
- Design of the ready queue
  - Centralized (one queue), distributed (one per processor), hierarchical (combination of both)

## Data affinity in multiprocessors

- A process can be scheduled on different processors during different quanta given to the process
  - Known as process migration
  - Each migration comes with the overhead of cache warm-up and possible remote memory accesses
  - A process can specify its affinity toward a processor through system calls
    - Prevents the scheduler from migrating the process to a different processor
  - A multiprocessor scheduler must be aware of data affinity and the overheads involved in migration

## Load balancing in multiprocessors

- Distributed and hierarchical ready queue designs may lead to load imbalance
  - Scheduler's responsibility to keep all processors equally busy
  - Receiver-initiated and sender-initiated diffusion (RID and SID)
    - Also known as pull migration and push migration
    - RID is invoked on a processor whose ready queue size has dropped below a threshold; migrates a number of processes from a processor whose ready queue size is above the threshold
    - SID is invoked on a processor whose ready queue size has gone above a threshold

## Load balancing in multiprocessors

- RID and SID algorithms can work together
  - In multiprocessor Linux, every 200 ms the SID algorithm is invoked on every processor and the RID algorithm is invoked whenever the ready queue of a processor is empty
- Load balancing and data affinity have conflicting goals
  - Achieving both is usually challenging

## Case study: UNIX

- Priority-based fixed quantum scheduling
  - Priority of a process may change during its life time
  - A higher priority value indicates lower priority
  - The timer interrupt handler keeps track of the CPU usage of the currently running process
  - Every one second, the priorities of all the user mode processes are updated by the timer interrupt handler
    - Set CPU usage of each process to CPU usage/2
    - Set priority value of each process to (base priority + CPU usage/2). The base priority is the minimum user mode priority value. Priority is inversely related to CPU usage.

## Case study: UNIX

- Priority-based fixed quantum scheduling
  - A process on entering the kernel mode receives a priority value lower than the user mode base priority indicating a higher priority than all user mode processes
  - A process about to enter the sleep state (in kernel mode) receives a predetermined priority value
    - Value depends on the reason to sleep
    - I/O calls from lower levels of the kernel receive very high priority because these calls hold a lot of resources
    - Disk I/O always receives higher priority than a process waiting for some memory resources
    - This priority is used to schedule the process right after it returns to the ready queue at the end of the sleep (still in kernel mode)

## Case study: UNIX

- Priority-based fixed quantum scheduling
  - A switch from kernel to user mode sets the process priority value to at least the user mode base priority
  - The priority of a process can be controlled by the owner of the process through the nice() call
    - Takes an integer argument, which is usually non-negative
    - The passed argument is added to the priority value of the process
    - The value of the argument is usually called the “nice value” indicating how nice the process is to the other processes
    - The nice value, the priority value, and the CPU usage value of a process are stored in the process table entry
    - Forked children inherit the nice value of the parent
  - Priority ties are broken by scheduling the process with a larger waiting time in the ready queue

## Case study: Linux

- Similar to UNIX with a few differences
  - Time quantum is not fixed and is inversely related to the priority value (i.e., lower priority processes get smaller time quantum)
  - The priority of a process is updated only after it gets at least one quantum to execute
  - Supports symmetric multiprocessing and each processor executes the scheduling algorithm locally
    - Does not honour global priority
  - An interactive process receives a nice value of -5 if it is sleeping on I/O for a long time
    - When such a process returns to the ready queue they get a somewhat higher priority to improve the degree of interaction

## Case study: Solaris

- Four classes of processes
  - Time sharing, interactive, system, and real-time
- A user process is classified as time sharing unless it is interactive (e.g., GUI process) or real-time
  - Multi-level feedback queue scheduling with queues ordered by priority; a lower priority queue can be scheduled only if all higher priority queues are empty
  - The processes in the highest priority queue get the smallest time quantum (can be switched quickly so that all high priority processes make some progress)
  - When a process returns from sleep state to ready state, its priority is boosted to somewhere between 50 and 59

## Case study: Solaris

- Interactive class executes a similar scheduling algorithm as the time sharing class
  - The GUI processes are assigned the highest priority
- System class includes all kernel processes
  - User processes executing in kernel mode are not in this class
  - Scheduler, system daemons, etc. are in this class
  - Each system process has a pre-determined fixed priority, based on which they get scheduled
- Real-time processes are time-critical
  - Assigned highest global priority

## Case study: Solaris

- At the time of selecting a new process for scheduling
  - The scheduler translates all local priorities within each class into global priorities
    - Done through a pre-determined priority order among the classes
  - Selects the process with the highest global priority
    - If there are multiple such processes, all of them are chained up in a single special queue
    - This special queue is scheduled in a round-robin fashion with a fixed pre-determined time quantum assigned to each process
  - A process is pre-empted if it goes to sleep state, or its time quantum expires, or a new higher-priority process arrives

## Case study: Windows XP

- Pre-emptive priority-based scheduling
  - 42 priority levels
  - Six priority classes: REALTIME, HIGH, ABOVE_NORMAL, NORMAL (this is default), BELOW_NORMAL, and IDLE
  - Seven relative priority levels within each priority class: TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL (this is default), BELOW_NORMAL, LOWEST, and IDLE
  - The global priority level of a process is determined based on its priority class and its priority level within the class
    - The Win32 kernel thread library allows the user to control these two parameters from the user program

## Case study: Windows XP

- Pre-emptive priority-based scheduling
  - The priority has an inverse relationship with CPU usage like in UNIX
  - A process returning to the ready queue from sleep state undergoes a priority boost
    - The amount of boost depends on the reason for sleep
    - A process sleeping on keyboard I/O always gets a bigger priority boost compared to one sleeping on disk I/O
  - Interactive processes use a different variable-quantum scheduling algorithm (next slide)

## Case study: Windows XP

- Pre-emptive priority-based scheduling
  - The process currently selected on the screen is called a foreground process and everything else is a background process
    - The foreground processes always get a bigger quantum than the background processes
    - When a background process moves to foreground, its quantum is multiplied by a constant positive integer larger than one (usually three)

## Scheduling in thread libraries

- Kernel-level threads are scheduled by the OS and the user-level threads are scheduled by the thread libraries
  - A group of user-level threads may be mapped to one or more kernel-level threads
- Process contention scope is used to carry out user-level thread scheduling within a process
  - OS scheduler maps the user-level threads to the available kernel-level threads
  - When a kernel-level thread is scheduled, the user-level threads mapped to it share the quantum through user-level thread switching or one of the mapped threads runs
  - User-level thread priorities are interpreted relative to the global priority

## Scheduling in thread libraries

- System contention scope
  - All threads are visible to the kernel
  - Thread priorities are interpreted relative to the process threads only
  - Linux supports only system contention scope
- Contention scopes in the POSIX thread library
  - pthread_attr_getscope() and pthread_attr_setscope() for getting and setting scheduler scope
    - Two defined scopes: PTHREAD_SCOPE_PROCESS and PTHREAD_SCOPE_SYSTEM

## Scheduling in thread libraries

- POSIX thread library allows one to specify the thread scheduling policy and thread priorities
  - Priorities are non-negative integers less than 100
  - The scheduling algorithm can be read and written to using the pthread_attr_getschedpolicy() and pthread_attr_setschedpolicy() functions
  - The thread priorities can be read and written to using the pthread_attr_getschedparam() and pthread_attr_setschedparam() functions

## Evaluating scheduling algorithms

- Must evaluate a set of algorithms before picking one
  - Need to decide the optimization criterion first
  - Possible example: maximize CPU utilization under the constraint that the maximum response time is T
  - Possible example: maximize system throughput under the constraint that the turnaround time of each process is linearly proportional to its execution time
    - Short-burst processes wait less
  - Once the criterion is decided, a set of candidate algorithms must be evaluated to select the best one

## Evaluating scheduling algorithms

- Analytical modeling
  - Design a mathematical model for the entire system that takes as input the description of the scheduling algorithm and the description of the processes possibly in terms of the distribution of arrival times, values of CPU and I/O burst lengths or simply a sequence of CPU and I/O bursts
  - The model computes the target criterion
  - Such a model can be as simple as a single formula or as complicated as a queuing model
  - Usually difficult to capture the real-world behavior of the system, but useful for eliminating some obviously poor scheduling algorithms

## Evaluating scheduling algorithms

- Simulation
  - Rigorous simulation must be carried out before selecting a few good candidates
  - A simulator is a software model of the system, but simpler than the full OS
  - The simulator can accept real-world user programs and simulate them to evaluate the target criterion
    - NachOS is a simplified OS simulator
- Final phase of the design involves incorporating the shortlisted candidate algorithms in the real OS and evaluating the target criterion