**OPERATING SYSTEMS LEC 04**

**Why Schedule the CPU**

* Maximise CPU utilisation
* Process execution consist of a cycle of CPU execution and I/O wait.
  + CPU and I/O burst cycle
* Process can be:
  + CPU-bound process: very long CPU burst
  + I/O-bound process: short CPU burst

**When do we switch CPU**

* From running to waiting state
  + I/O request, call wait
* Running to ready state
  + timer off
* Process terminates

**CPU Schedulers**

* The CPU scheduler selects a process in memory ready for execution and allocates the CPU to it.
  + This is called a short-term scheduler.
  + Can be a FIFO, priority, tree or unordered linked list with PCS of processes stored.
* Other schedulers include:
  + Medium Term Scheduler: Swaps job in/out of memory to reduce contention
  + Long Term Scheduler: Determines which jobs are admitted
    - Invoked when a job finished (less frequent than short term)
    - Controls degree of multiprogramming
    - May be absent on time-sharing systems (MTS instead)

**Preemption**

* Non-Preemptive: Once CPU has been allocated to a process, it keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
* Preemptive: CPU can switch during when a process switches from running to ready and also waiting to ready.
  + Preemptive requires special hardware – such as timer.
  + Can result in race condition
    - When a process updating share data is pre-empted and the second process is accessing the data
  + Careful when pre-empting a kernel process.

**Dispatcher**

* Module that gives control of CPU to the process selected by short-term scheduler
* Involves:
  + Switching context
  + Switching to user mode
  + Jumping to proper location in user program to restart program
* Dispatch latency: time taken to stop one process and start running another
  + Must be short as every context switch invokes dispatcher

**Scheduling Criteria**

* Criteria used to compare scheduling algorithms to determine which is better
* CPU Utilisation
  + Percentage usage of CPU – higher is better
  + Throughput – number of processes that complete their execution per time unit – higher the better
  + Turnaround time – amount of time to execute a particular process – shorter the better
  + Waiting time – sum of periods a process spent waiting in ready queue – shorter the better
  + Response time – time taken from when a request was submitted until the first response is produced, not output – for time sharing environment.
* For interactive systems:
  + Minimise variance in response time – easier with predicable response.
  + Minimise maximum response time OR average response time.
* How about fairness?
  + ie. Does each process get a fair share of CPU time
* waiting time = turnaround time – burst time

**First Come First Served (FCFS) Scheduling**

* A process that requests CPU first is allocated CPU first
* Easy implementation with FIFO queue
* Non-Preemptive
* Performance: poor waiting, turnaround AND response time.

**Shortest Job First (SJF) Scheduling**

* Convoy Effect: short processed behind long process.
* Associate each process with the length of its next CPU burst, using the length to schedule the process with the shortest time.
* Two Schemes:
  + Non-Preemptive – once CPU is given to process it cannot be pre-empted until it completes its CPU burst
    - Typically has a higher turnaround time and longer wait time
  + Preemptive – If new process arrives with CPU burst length less than remaining time of current executing process, pre-empt.
  + Shorter wait time and faster turnaround time.
* SJF is optimal – gives minimum average waiting time, turnaround time and response time for a given set of processes.
* Moving short job before long one decreases waiting time for short job more than it increases the waiting time of the long job
  + average waiting time decreases
* Only optimal when all jobs are available simultaneously

A screenshot of a cell phone

Description automatically generated**Length of next CPU Burst**

* Can only estimate the length
* Use exponential averaging.

**Priority Scheduling**

* A priority integer is associated with each process
* CPU is allocated to process with highest priority (low number = high priority)
  + Preemptive – pre-empt CPU if priority of arrived processes is higher than the priority of current running process.
* Internally defined priority – based on measurable quantity
  + Time limits, memory requirements, no. of open files
* Externally defined priority
  + Type/amount of funds, department, politics
* Problem: Starvation – low priority processes may never execute
  + Solution: aging – time progressing increases priority of process.
* Shortest Job first is priority scheduling – priority predicted by next CPU burst time

**Round Robin (RR) Scheduling**

* Each process gets small unit of CPU time (time quantum) usually 10-100 ms.
  + After the time, process is pre-empted and added to end of ready queue.
* If ready queue has n processes and time quantum is q.
  + Each process gets 1/n of CPU time in chunks of q time units.
  + No process waits more than (n-1) q time unit
* Performance of this scheduling depends on size q.
  + Large q – FCFS
  + Small q – q must be large with respect to context switch
    - Otherwise overhead is too high
  + Turnaround improves if most processes finish next CPU burst within q.
  + 80% of CPU burst is at most q.
* Typically, higher average turnaround than SRTF but better response time.

**Multilevel Queue Scheduling**

* Two separate queues – foreground (interactive), background (batch)
  + Each has own algorithm
* Scheduling occurs between queues
  + Fixed priority scheduling – serve all from foreground then from background
    - Starvation issue possible
  + Time slice – each queue gets a certain amount of CPU time.
    - Eg. 80% to foreground in RR and 20% to background in FCFS.

**Multilevel Feedback Queue Scheduling**

* A process can move between queues.
  + One form of aging mechanism to prevent starvation.
* These schedulers are defined by the following:
  + Number of queues
  + Scheduling algorithm for each Queue
  + Method used to determine when to upgrade a process
  + Method to determine when to demote a process
  + Method for determining which queue a process will enter when it needs service.

A screenshot of a cell phone

Description automatically generated

**Guaranteed Scheduling**

* Make real promises to user about performance and live up to them.
  + Eg. for *n* users, the promise can be 1/*n* CPU time for each user.
* System keeps track of CPU time per user for all processes since login and how long the user has been logged in.

**Multiple-Processor Scheduling**

* Homogenous Processor – all processors are identical in a multiprocessor
* Load sharing:
  + Queue for each processor – load not balanced
  + A single ready queue for all processes, to approaches:
    - Each process is self-scheduling – need mutual exclusion on the ready queue; symmetric multiprocessing
    - Master-slave structure – one processor as scheduler, asymmetric multiprocessing.
* Asymmetric multiprocessing – only one processor accesses system data structures
  + No need for data sharing
  + Simpler system

**Real-Time Scheduling (Hard)**

* Hard real-time systems require to complete a critical task within a guaranteed amount of time
  + Each submitted process has a statement of the amount of time needed to complete/perform I/O
  + The scheduler:
    - Admits process if it can guarantee process will complete on time
    - Rejects the process if impossible
* Should know how long each type of OS function takes to perform
  + Impossible in a system with secondary storage or virtual memory
* Composed of special purpose software running on hardware dedicated to their critical process.

**Real-Time Scheduling (Soft)**

* Requires critical process to have priority over others
* General-purpose systems supporting (ie) multimedia, high speed interactive graphics
* Requirements:
  + System must have priority scheduling
    - Real time processes always have highest priorities.
  + Dispatch latency must be small, we do this by:
    - Allow system calls to be preemptible
    - Make kernel preemptible – create priority inversion: high priority process waits for lower ones to finish
    - Solution: Use priority inheritance where a low priority task using resources needed by higher priority task inherits its priority until completing the resources.

**Dispatch Latency**

* Two components of conflict phase
  + Preemption of kernel process
  + Release of low-priority resources that are needed by a high-priority process

**User-Level Thread Scheduling**

* Thread library schedules user level threads to run on an available kernel label thread
  + Process-contention scope (PCS)
  + Many-to-one and many-to-many models
* Priority scheduling is commonly used
* Thread is not running in CPU yet
  + Kernel doesn’t know of existence of user-level thread
  + Context switch (kernel) when time quantum for process is up.

**Kernel-Level Thread Scheduling**

* Kernel schedules which thread gets CPU
  + Threads can be from same or different processes
  + Known as system-contention scope (SCS)
  + For one-to-one model use only. Linux, Windows XP
* Each thread is given time quantum, context switch is for each thread
* Context switch in kernel thread is more expensive than for user-level thread
* Context switch between threads from same process is faster than different processes.

**Deterministic Model**

* Takes a particular predetermined workload and defines performance of each algorithm for that workload
* Simple, fast
* Requires exact numbers for input
* Results are indicative only for this input
* Too specific to be of general use

**Queueing Model**

* Average queue length, *n*
* Average waiting time in queue, *w*
* Average arrival rate to the queue: *λ*
* System steady state – number of processes leaving the queue must be equal to number of processes that arrive thus:
  + Little's formula: *n =* λ *\* w*
* Useful for:
  + Comparing algorithms
  + Only as good as distributions
    - Often difficult and unrealistic to make problem mathematically tractable

**Simulations**

* Involves programming a model of computer system
* Software data structures represent major components
* Variable representing the clock
* Data can be generated by:
  + Random number generator
  + Trace tapes – created by monitoring real system
  + Simulation can be very expensive
    - Hours of CPU time, large amount of storage (trace tapes), time for designing, coding and debugging.

**Implementation**

* Only completely accurate way to evaluate a scheduling algorithm.
* Code it, put it in OS, see how it works
* Difficulties:
  + Cost of the approach
  + Environment in which algorithm is used may change.

**Linux 2.2**

* Has two separate process-scheduling algorithms:
  + Timesharing – fair Preemptive
  + Real time – absolute priorities
* User-mode processes can be pre-empted (only these)
* Processes have scheduling class containing a prioritised, credit-based algorithm
  + Each process has a number of credits – larger means highest priority
  + Credit is decremented by one each time a timer interrupt occurs – process is suspended when it becomes 0
  + If no runnable processes, do re-crediting to every process in the system
    - credits = credits/2 + priority.
    - O(n) where n is total number of processes.
  + High priority to I/O bound processes
* Real time Scheduling: FCFS, RR

**O(1) Scheduling**

* Problem with Linux 2.2 is it gets slower when total number of tasks increase. O(n)
* Scheduling algorithm in Linux 2.5 is O(1), providing better support for SMS systems
  + Runs in constant time, irrespective of total number of processes, n
* Preemptive and priority based
* Two separate priority range (lower value has higher priority)
  + Real time: 0-99
  + Nice: 100-140
* Higher priority task is assigned with higher time quantum
* Kernel maintains list of runnable tasks in queue that contains two priority arrays
  + Active: all tasks with time remaining in their time slices
  + Expired: All tasks with no remaining time in their time slices
* Scheduler selects highest priority task in active array
* For SMP, each processor maintains own runqueue and schedules itself independently.
* Recalculates new priority of each process with exhausted time and move it to expired.
* Calculate new priority:
  + Real time task with fix priority
  + All other tasks have dynamic priorities (calculated before going into expired)
    - *nice* value + *d*; where -5 ≤ *d* ≤ +5
  + Value for *d* is determined by task interactivity – how long process has been sleeping, waiting for I/O
    - Longer sleep – more interactive – *d* is set close to -5
  + CPU bound process – gets lower priority
* When active queue is empty – swap active and expired

**Completely Fair Scheduler (CFS)**

* O(1) has poor response time for interactive processes
* Kernel release 2.6.23 makes CFS default for Linux
* Linux is based on scheduling classes
  + Each class has a specific priority
  + Next task to run is task with highest priority in the highest priority class
* Two scheduling class primarily (can add more)
  + Default class using CFS scheduler
  + A real time scheduling class
* CFS assigns a proportion of CPU time to each task
  + Calculated based on the nice value (-20 to +19) assigned to each task
    - Lower means higher priority – higher proportion of CPU time
    - Default nice value is 0
* CFS uses targeted latency – an interval of time where every runnable task should run at least once
  + Has default and minimum values
  + Increases when the total number of tasks in the system increases above a threshold value
  + Proportions of CPU time are allocated from the value of targeted latency
* CFS keeps virtual run time (vruntime – how long a task has run) for each task
* Vruntime is associated with decay factor based on priority
  + Lower priority has a higher decay factor
  + Normal task (nice value 0) has vruntime equal to physical runtime
    - normal priority that runs for 200 millisecond has vruntime. = 200 milliseconds.
    - Lower at 200 milliseconds has vruntime > 200.
    - Higher at 200 milliseconds has vruntime < 200 milliseconds
  + CFS selects task with smallest vruntime to run next
    - Higher priority task can pre-empt a lower one