Vika 7

TÖL401G - Stýrikerfi

1. Motivation for scheduling

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

2. Multiprogramming (Batch) system

Maximise CPU utilisation and throughput of jobs. While one process is blocked (i.e. due to I/O) another process may use the CPU.

3. Timesharing (Multitasking) system

While one process/thread is performing calculations, user can still interact with another process/thread because scheduler switches often between them.

4. Why Scheduling is Reasonable

Overhead of scheduling is generally outweighed by the benefits. Process execution typically consists of a period of CPU usage and subsequent I/O wait. During this wait scheduling enables another process to utilise the CPU.

5. Definitions

- (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, switching from kernel to user mode)
- Scheduling algoritm The algorithm used by the scheduler to decide which process/thread gets the CPU for how long.

6. Remarks

- There is no **best** scheduling algoritm. Different algorithms are best suited to different types of system (batch, multitasking, realt-time, etc.) and usage scenarios.
- In operating systems with kernel level threads, only threads (not processes) are scheduled.

7. Preemptive vs Non-preemptive scheduling

7.1. Non-preemptive

CPU is allocated to one process until that process blocks or terminates. Timesharing is only possible if CPU bound processes explicitly yield the CPU, by using the yield system call to voluntarily transition from **RUNNING** to **READY** state.

7.2. Preemptive

A running process may be interrupted at any time because its time slice has expired. The scheduler then takes control and determines which process gets to use the CPU next. This may be the same process, or any other process in **READY** state.

7.2.a. Potential problems

- A process may be interrupted while updating data shared between processes. Cooperating processes must therefore synchronise access to such resources to ensure a single process has exclusive access to the data until it completes updating it.
- A time slice timer can expire while kernel code is being executed, the kernel must therefore disable interrupt processing while updating critical kernel data structures.

8. Scheduling Criteria

All scheduling systems must ensure:

- Fairness
  Each process gets CPU time
- Enforcement of priorities
  High priority processes are preferred.
- Balance
  All the different resources of a system are reasonably utilised.

8.1. Batch Systems

Batch system schedulers must also ensure:

- CPU utilization
  The CPU should be kept as busy as possible.
- Throughput
  The number of processes that complete their execution per time unit should be maximised.
- Turnaround time
  Minimise the amount of time (from start to termination) to execute a particular process.

8.2. Timesharing/Multitasking systems

An interactive system must ensure:

- Response time
  Minimise the amount of time it takes from the time a request is submitted until the first response is produced.

8.3. Real-Time systems

A real-time system must ensure:

- Meeting deadlines
  Processes (or events within processes) that must be started/finished before a certain point in time must be preferred.
- Predictability
  As long as the system is not overloaded, it can be predicted when a certain process (or event within) is executed.

9. Scheduling Algorithms

FCFS, SJF, and SRTF are primarily applicable to Batch operating systems, since each process runs more or less to completion. To present the illusion of multiple processes running simultaneously Interactive operating systems must employ different algorithms, such as Round Robin and its variants.

9.1. First Come First Served (FCFS)

In a FCFS scheduler processes are allocated CPU time in the order of arrival, and running processes are not interrupted. This means that the first process to arrive will run to completion, before the second gets the CPU and runs to completion, etc. This algorithm is non-preemptive, easy to

implement, and fair (in the sense that all processes will eventually get access to the CPU). For this reason the average waiting time for a process, and the general suitability of the algorithm are heavily dependent on the order in which processes are created.

9.2. Shortest Job First (SJF)

In a SJF scheduler processes are served in ascending order of CPU time required (based on the processes in queue at the time of scheduling decissions). This imposes the limitation that the CPU time required by a process must be known in advance (unlikely in real world scenarios). This algorithm suffers from the problem that it is unfair, since a process requiring large amounts of CPU time will never be executed if shorter processes keep arriving.

9.3. Shortest Remaining Time First (SRTF)

The SRTF algorithm is a preemptive variant of SJF, where the process running process is interrupted if a newly arrived process requires less CPU time than the running process would require to complete. This algorithm still suffers from the unfairness problem of SJF, where a long process will never get the CPU if shorter processes keep arriving.

9.4. Round-Robin (RR)

In a Round-Robin scheduler CPU time is divided into **time slices** with a fixed maximum duration (If a process completes before its time slice expires, the next process does not wait for the time slice to expire, but starts immediately.) Processes are then served in a First Come First Served manner (with new processes simply placed at the back of the READY queue), with each process getting the CPU for one time slice, before being placed at the back of the queue. When a process terminates it is removed from the READY queue. If a process blocks, i.e. due to I/O it is removed from the READY queue and placed onto the WAITING queue until its blocking request has been satisfied, at which point it reenters the READY queue.

9.5. Round-Robin with Priorities

In a Round-Robin with Priorities scheduler the READY queue is replaced by multiple queues, where each queue has a priority value. The highest priority queue is processed in a Round-Robin fashion, and only once it is empty is the next queue processed. This has the potential to cause starvation in low priority processes, which can be countered by dynamically adjusting the priority of processes (Increase the priority of processes that have spent a long time waiting, decrease the priority of long running processes).

9.6. Multilevel Queue Scheduling

Different categories of processes (interactive, background, system, etc.) are placed in different queues. Each queue has a different scheduling algorithm. Some sort of algorithm is required to choose which queue gets to run.

9.7. Multilevel Feedback Queue Scheduling

Multilevel Queue Scheduling, except processes can be moved between queues.

10. Thread Scheduling

If user level threads are used, the OS kernel is not aware of the existence of the threads, but simply schedules the processes. Scheduling of the threads is left to the user level thread library. If kernel level threads are used the kernel schedules threads, and typically does not care to which process those threads belong.

11. Multiple-Processor Scheduling

When more than one CPU/core are present in a system, and share memory, each core must be managed by the operating system.

11.1. Asymmetric multiprocessing

Only one **master** processor/core accesses the kernel data structures (such as scheduler queues). Other processors (**slaves**) wait for the master processor to assign them work.

11.2. Symmetric multiprocessing (SMP)

All processors/cores run the same kernel, and make independent scheduling decisions. This is the scheme used by all major operating systems these days. This can either be implemented by a shared scheduler queue, access to which must then be synchronised, or each processor can maintain its own scheduler queue.

11.3. Processor Affinity

Since each CPU core has its own cache for recent data and instructions it is inefficient to constantly move processes between cores, and thereby invalidate all caches, requiring costly memory accesses, the scheduler tries to keep a process on the same physical core. The process is then said to have affinity for that processor.

11.3.a. Hard processor affinity

In a hard affinity model a process is never moved between processors, such as when each processor has its own scheduler queue. Under this model some cores may sit idle, even though processes are waiting in queue, because they have an affinity for a different processor.

11.4. Load Balancing

In opposition to Processor Affinity, load balancing attempts to evenly distribute workload between available processors.

11.4.a. Push migration

In a push migration scheme the kernel periodically checks the load on each processor and migrates (pushes) processes from cores with high load, onto cores with light load.

11.4.b. Pull migration

In a pull migration scheme a processor whose scheduling queue is empty will pull processes form another processor's queue.

11.4.c. Soft processor affinity

Load Balancing and Processor Affinity contradict each other, and it is difficult to develop algorithms that achieve a good compromise between the two. Such attempts are known as soft processor affinity, and revolve around attempting to maintain affinity, but allowing load balancing where necessary.

12. Memory Stalls

On the OS level processes get blocked while waiting for things like I/O. The same may happen on the CPU level, since main memory access is significantly slower than the CPU itself. When this occurs it is known as a memory stall, and results in wasted CPU cycles. Memory Stalls are counteracted by larger CPU caches, and hyper-threading.

13. Hyper-threading

Hyper-Threading, also known as Hardware multihreading, Simultaneous Multithreading, or Chip multithreading involves a CPU core presenting itself as two cores. In reality there is only one core, capable of switching between two threads of execution in case of a memory stall, i.e. if thread 1 stalls the CPU starts executing thread 2. In some cases even the OS may not be aware of hyper-threading, which may cause problems on a multi core system, where a scheduler may in theory schedule two processes to run on logical cores belonging to the same physical core, leaving one core running two processes and the other core idle. This problem is solved by making the OS aware of hyper-threading.