Exercise 1

Processes, Threads and scheduling

What is an Operating System?

- No one absolute definition but it fulfils different purposes
  - Execute user and system programmes
  - Abstract usage of hardware => Convenience of use
  - Assure efficient use of hardware => Resource allocation and management
  - Provide a user interface (CLI, GUI, buttons, etc.)
  - Offer file manipulation and communication services
  - Error handling => Avoid black screens

=> Goals depend on machine, i.e., a desktop office computer creates other needs than a microchip in a TV

Components of an operating system

- Kernel:
  - Always Running when computer is on
  - Loaded into memory and executed on boot
  - E.g., I/O control, process scheduling, memory management
- System applications
  - Not part of the kernel => Only executed when needed
- Middleware:
  - Additional, commonly necessary, services for app developers
  - E.g., Database software, compilers, runtime environments

Interrupts

- I/O devices run slower than CPU and memory => When I/O is finished, the device informs OS and programme can continue execution => Interrupt driven OS
- Hardware interrupt: Caused by I/O device
- Software interrupt: Software error or system call (= Application needs an OS service)
  - State of CPU is saved (Programme counter and register contents)
  - Interrupt handling scheme is run => Depends on OS and type of interrupt
  - System calls are usually invoked via an OS specific API

Dual mode operation

- Some actions are privileged to avoid damages to the system => Can only be executed by the OS
- Kernel mode
  - Privileged code that can only be invoked by OS
  - Started by interrupts, reset on return from interrupt
- User mode
- More modes are possible, depending on OS architecture

A few terms

- Single processor vs. Multi-processor systems

- Single core vs. multiple cores (=> Computational unit)

- Multiprogramming => Multiple programmes are kept in memory at the same time (= are in execution)

- Time-sharing => Execution changes among programmes in execution to create impression of simultaneous execution

Process

- A programme in execution
  - Loaded into memory
  - Executed by CPU, waiting for I/O or waiting for CPU time
- OS manages execution of multiple processes
- Program counter => Specific CPU register containing the memory location of the next instruction to be executed

Process memory

- Text section => Executable application code (fixed on creation)
- Data section => Global variables (fixed on creation)
- Heap section => Dynamically allocated memory
- Stack section => Temporary data storage, e.g., function parameters

=> A programme has only text section, a process has them all

Process states

- New => Process is created
- Running => Instructions are being executed (has CPU time)
- Waiting => Process is not executed but waiting for an event, e.g., I/O
- Ready => Process is waiting for CPU time
- Terminated => Process has finished (will be removed from memory)

Process control block

- Contains information about and current state of a process
  - Process state
  - Program counter and contents of CPU registers
  - CPU scheduling, memory management, I/O status, and accounting info

- Used for context switching
  - CPU can switch between processes for better performance
  - When a process enters the CPU, state is read from PCB
  - After execution in CPU new process state is saved in PCB
  - Introduces time overhead

Inter-process communication

- Processes may be cooperating or be dependent on one another
  - Information sharing
  - Multiple computers working on one problem / Modularity => Speedup
- Shared memory
  - Multiple processes have access to the same memory space
  - Fast, but high demands on application programmers to avoid errors
- Message passing
  - OS offers services to send and receive messages among processes
  - Slower, but less error-prone than shared memory

Process scheduling

- Goal is to maximize CPU utilization and to void CPU idle time
- CPU scheduler chooses a waiting process and allocates it to a CPU
- Non-preemptive scheduling => Processes are switched only when a process goes into waiting state or terminates
- Preemptive scheduling => Processes can be switched according to additional rules (depending on scheduling algorithm)

Scheduling criteria

- CPU utilization => Should be as high as possible

- Throughput, turnaround time, waiting time, response time

- Generally, goal is to keep waiting and response time low, but priority of criteria depends on system and tasks

Scheduling algorithms

- First-Come, First-Serve (FCFS)
  - Every new process is placed in a queue
  - On change to wating or termination the next in the queue executes
  - Convoy effect => Short processes behind a long process result in a high average waiting time
  - Simple, but potentially slow regarding average waiting time

Scheduling algorithms

- Shortest-Job-First (SJF)
  - Each process has information about the length of its next CPU burst
  - Preemptive or non-preemptive
  - Shortest burst time gets scheduled first
  - Optimal regarding average waiting time and short turnaround time
  - Approximation of length of next CPU burst is necessary => Adds complexity
  - Estimation done via the lengths of previous bursts

Scheduling algorithms

- Priority scheduling
  - Priority number is allocated to each process
  - Process with highest priority is chosen
  - Preemptive or non-preemptive
  - If multiple processes have the same priority another scheme is used to decide, e.g., FCFS
  - SJF is a form of priority scheduling where priority depends on next CPU burst time
  - Can introduce starvation of low priority processes => Aging can be introduced where priority is increased with time in the ready queue

Scheduling algorithms

- Round Robin scheduling
  - CPU time is split into time quantums (typically 10-100 ms)
  - Processes are added to ready queue as they arrive
  - When process terminates, goes into waiting state or quantum is over the next process in the queue is loaded
  - If work is remaining for a pre-empted process, it is enqueued again in the ready queue
  - Higher turnaround time than SJF but better response times
  - Quantum must be long compared to context switch time
  - If quantum gets too large Round Robin becomes FCFS

Multilevel queues

- Multiple scheduling queues can be used in parallel to optimize their strengths
- Each queue has its own scheduling algorithm
- E.g., division into foreground (interface) and background processes
- Scheduling is necessary between queues

Threads

- A process can be split into multiple threads of execution => Multiple tasks can be done at the same time
- Each thread has a unique id, a program counter a register set and a stack
- Thread creation is more lightweight than process creation => Threads share resources of their invoking process (data, files, code)

Threads vs Processes

- Benefits of threads
  - Improve responsiveness of user interfaces => Programme does not freeze while a computationally intensive task is done
  - Simple resource sharing among threads => Share data of process
  - Less overhead for thread switching compared to context switching
  - Scalability => Application can easily make use of multicore architectures

- Drawbacks of threads
  - If one threads crashes, the whole process crashes
  - Synchronization must be implemented by developer for data consistency

Parallelism vs Concurrency

- Parallel system can execute several tasks simultaneously, i.e., multiple processes can be in execution at the same time in different CPU cores

- Concurrency means that multiple tasks can make progress aside one another, regardless of the number of CPUs and cores

=> Concurrency can make use of a parallel system but can also take place in a single-core system

User threads vs kernel threads

- Kernels in most cases are multithreaded where the threads of execution lie in kernel space
- Application threads are executed in the user space
- Mapping is necessary between user threads and kernel threads
  - Many user threads to one kernel thread => Little overhead but one blocking user thread can block all other threads
  - One user thread to one kernel thread => Overhead for thread creation but user threads don´t block each other (Windows, Linux)

Hyperthreading

- Each physical CPU core is divided into virtual cores
- Virtual cores are available to the OS in the same way as physical ones
- Can speed up system efficiency and is commonly used in contemporary CPUs

Questions?

Processes, Threads and scheduling