Overview

Thread Programming

- The programming of **multicore processors** is closely coupled to **shared memory** parallel programming and **thread programming**.
- **Threads**:
  Computation streams of the a program that are computed in parallel and that access common variables of the shared memory.
- Popular thread programming environments:
  - Pthread, Java threads, OpenMP and Windows threads

Overview

Processes

A **process** is a program in execution and includes the program code and all information necessary for the execution (e.g. data on the runtime stack or on the heap, the content of the registers).

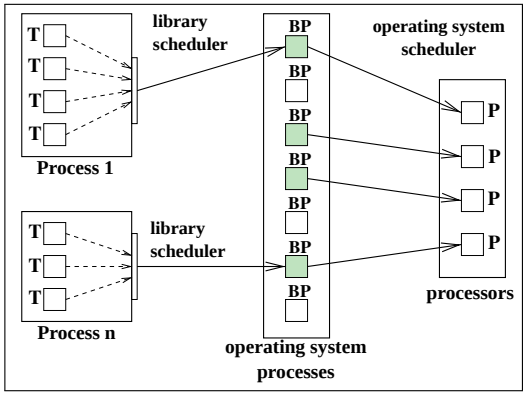
- Private address space
- Context switch: Change of state due to process switching
  - Multitasking: Processes are executed in a time slicing method
    $\rightsquigarrow$ Concurrent but not parallel
  - Multiprocessing: true parallelism
- Creation of a process: `fork` call
  Child process: *copy* of the address space, identical program after the `fork`-call (except for return value of `fork()`)
- Creation and management of processes is relatively time-consuming

Threads

- Extension of the process model: A process includes several independent computation streams (threads)
- Threads share the address space of their process
  ⇝ Fast information exchange (compared to sockets on process level)
- Fast creation of threads
- Execution on different processors or cores possible
  ⇝ True parallelism
- Different thread levels:
  - User-level threads: Fast thread switching; OS has no knowledge of threads
    ⇝ CPU stops whole process (e.g. for I/O); switching to another thread is impossible
  - Operating-system level

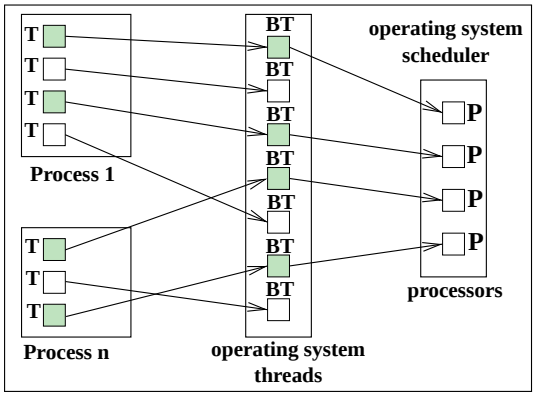
Execution Models for Threads –1–

N:1-Mapping



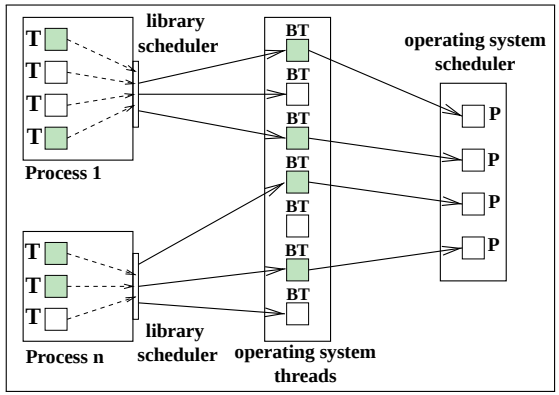
Execution Models for Threads –2–

1:1-Mapping



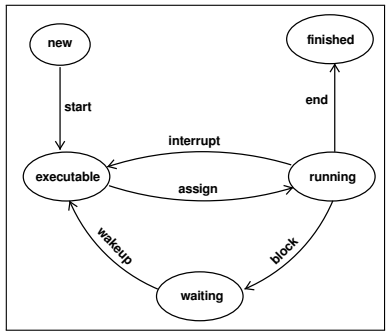
Execution Models for Threads –3–

N:M-Mapping



Thread States

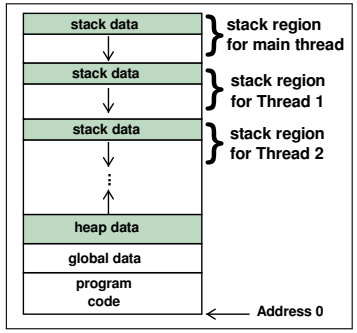
- New
- Executable
- Running
- Waiting
- Finished



Visibility of Data

Run-time management for a program with several threads

- Private runtime stack per thread for local data
- Local data of a thread is invisible for other threads
- Run-time stack exists as long as the thread exists
- Special methods for return value passing required



Overview

Synchronization = Coordination of Threads

- Achieving a desired order of execution
- Designing the access to the shared memory

- The order of execution of computations causes a special state of the shared memory, i.e. assignment of values to the shared variables.
- Avoidance of undesired effects during the parallel access to a shared variable

Effects of cooperation and coordination of threads can differ between concurrency or parallelism

Terms of Synchronization

- Barrier synchronization
- Race conditions
- Critical section
- Lock mechanism
- Condition variables
- Semaphor mechanism
- Monitor

Barrier synchronization:
All participating threads wait for each other and no thread executes an instruction that comes after the synchronization instruction until all threads have reached the barrier.

Critical Region

- **Race conditions:**
  The result of the parallel execution of a program fragment by several threads depends on the relative execution speed of the threads:
  - Executing the program fragment first by Thread T_1 and then by Thread T_2 may compute a different result than if first Thread T_2 executes the fragment and afterwards Thread T_1.

  $\rightsquigarrow$ **non-deterministic behavior**
- **Critical section:**
  Program fragment with access to shared variables that may also be accessed concurrently by different threads
  - A correct execution order can be achieved by using a **mutual exclusion**, to ensure that one thread exclusively can access the variable in the critical section.
  - Methods: lock synchronization, condition variables.

Locks

- **Lock variable** s (or mutex variable)
- Function `lock(s)`:
  Before entering the critical section, the thread executes `lock(s)` for locking the lock variable *s*;
- Function `unlock(s)`:
  After leaving the critical region, `unlock(s)` has to be called.

$\rightsquigarrow$ Only if *each* processor uses this convention, race conditions are avoided.

Effects of `lock(s)` for Thread T_1:

- If no other thread has locked *s*:
  Assignment of *s* to T_1
- Another Thread T_2 locks the lock variable *s*:
  Thread T_1 remains blocked until Thread T_2 releases *s* by calling `unlock(s)`

Condition Variables

- With **condition variables**, a thread T_1 blocks until a specific condition occurs.
- Waking up a thread can only be done by another Thread T_2.
- After waking up, thread T_1 tests the condition again.
- For avoiding race conditions, additional lock variables are used.

Semaphore Synchronization, Monitor

Semaphore synchronization:

- Structure that includes an integer variable s with two atomic operations
  - $P(s)$ (or $wait(s)$) waits until the value of s is greater than 0, decrements the value of s by 1 afterwards and allows the execution of the following statements.
  - $V(s)$ (or $signal(s)$) increments the value of s by 1.
- Typical pattern:

```
wait(s)
critical section
signal(s)
```

Monitor:

- Language construct that combines data and operations which access that data together into one structure
- Access to monitor data only using monitor operations
- At the same time, the execution of only *one* monitor operation is allowed

Overview

Problems of Thread Programming

Depending on the application, synchronization may result in a complex interaction of threads which might cause problems.

- Performance decrease due to sequentialization
- Deadlocks
- Memory access latencies and cache effects

Number of Threads and Sequential Execution

To achieve efficiency:

- An adequate number of threads has to be used
  - Sufficient parallelism versus thread creation overhead
- Sequential execution should be avoided:
  - Frequent synchronization leads to inactive threads

Deadlock

- **Deadlock**:
  State in which each thread is waiting for an event that has to be triggered by another thread (which also waits for an event unsuccessfully).

⇝ **Cycle of mutually waiting threads** occurs.

- Example:
  - Thread T_1 tries to lock the lock variable s_1 and then lock variable s_2; after locking s_1, thread T_1 is suspended;
  - Thread T_2 tries to lock s_2 and afterwards s_1; after locking s_2, thread T_2 is suspended;
- Both threads wait for the unlocking of the missing lock by the other thread, but the unlock will never be performed.

Memory Access Latency and Cache Effects

- Memory access latencies may require a large portion of the parallel runtime of a program.
- Dependencies between memory accesses of different processor cores:
  - Read-read dependencies: due to caches, possible without memory access
  - Read-write dependencies ⇝ memory access
  - Write-write dependencies ⇝ memory access
- Better: Design of the program, so that cores access **different** data
- **False Sharing**:
  Two different threads access different data items which are located in the same cache line
  ⇝ Memory operation
  Hard to influence

Overview

Arrangement of Threads

For organizing the cooperation of threads of a program **parallel design patterns** (=special structures of coordination of the threads) can be used.

- Creation of threads
- Fork-join
- Parbegin-parend
- SPMD and SIMD
- Master-slave or master-worker
- Client server model
- Pipelining
- Task pools
- Producer-consumer threads

Creation of Threads – Fork-Join

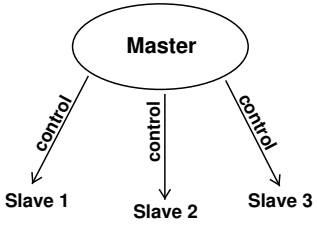
- Creation of threads
  - At the beginning of the execution there is usually only one thread
  - Static creation: A *fixed* number of threads is created at the beginning of the execution of the parallel program. The threads exist during the whole runtime and are destroyed at the end of the program.
  - Dynamic creation: Creation of threads at each time possible
- Fork-join
  - An existing Thread T_1 creates a second Thread T_2.
  - Arbitrary nesting
  - Specific characteristics in different parallel programming languages and environments

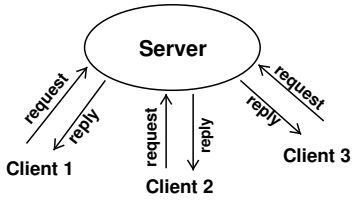
Parbegin-Parend – SPMD and SIMD

- Parbegin-parend
  - Simultaneous creation and destruction of several threads (structured variant of thread creation)
  - The statements included by Parbegin-Parend are assigned to separate threads;
    The statements following the Parend statement are executed after all additional threads are destroyed.
  - Actual parallel execution depends on implementation.
  - Specific characteristics in individual programming languages and environments (e.g. parallel sections in OpenMP)
- SIMD (***S**ingle **I**nstruction, **M**ultiple **D**ata*) and SPMD (***S**ingle **P**rogram, **M**ultiple **D**ata*)
  - All threads execute the same program but with different data
  - SIMD: *synchronously*, i.e. all threads execute the same instruction simultaneously. This means *data parallelism in the strict sense*.
  - SPMD: *asynchronously*, i.e. at a time different threads execute different program statements at the same time.

Master-Slave or Master-Worker – Client-Server-Model

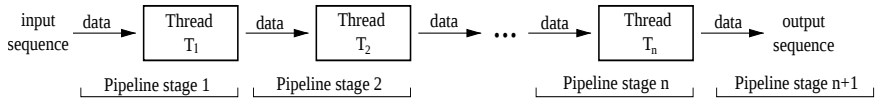
- Master-slave: One single thread controls the whole computations of a program; creates mostly similar *worker* (or *slave*) *threads* to which computations are assigned.
- Client-server model: Several client threads make requests to the server thread; server thread handles client requests concurrently/parallel
  (Extensions: Several server threads or threads which are client and server at the same time)





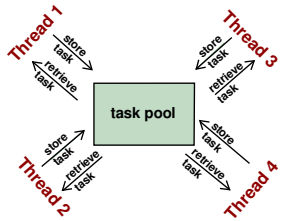
Pipelining

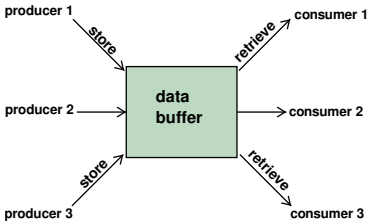
- Threads $T_1, \ldots, T_p$ are logically ordered in a specified order;
- Thread T_i gets the output of thread T_{i-1} as input and computes its output that will be used by threads T_{i+1}, $i = 2, \ldots, p-1$, as input;
- Thread T_1 gets its input from other program parts; T_p provides its output to other program parts
- Parallelism despite of data dependencies



Task Pools – Producer-Consumer

- Task pool: **Data structure** managing program parts in form of **functions** (tasks) that have to by executed; execution by a fixed number of threads that access the taskpool for extraction and storage of tasks.
- Producer-consumer: Producer threads create data and consumer threads use the data;
  common **data structure** of fixed size for the storage of **data**.





Overview

Parallel Programming Environments

- POSIX Threads (Pthreads):
  Portable thread library, implementations available for many platforms, standard API for Linux, Pthreads-win32 for Windows; programming language: C (other bindings available).
- Windows Thread API:
  C/C++-Environment for developing multi-threaded Windows applications.
- Java Threads:
  Creation, management and synchronization of threads on language level, provides special classes and methods, package `java.util.concurrent`
- OpenMP:
  API for portable multithreaded programs, Fortan, C and C++ platforms independent set of compiler pragmas and directives, special functions and environment variables