Methodology for Process and Image Computation.

Amlal El Mahrouss
amlal@nekernel.org

December 2025
Last Edited: January 2026

Abstract

CoreProcessScheduler governs how the scheduling backend and policy of the system works, It is the common gateway for schedulers inside NeKernel based systems.

1 I: Introduction.

CoreProcessScheduler (now referred as CPS) serves as the foundation between the scheduler backend and system. It takes care of process life-cycle management, team-based process grouping, and affinity-based CPU based allocation to mention the least.

1.1 II: The Affinity System.

Processes are given CPU time affinity hints using an affinity kind type, these hints help the scheduler run or adjust the current process.

1.2 Illustration: The Affinity Kind.

The following sample is C++ code. The smaller the value, the more critical the process.

```
enum struct AffinityKind : Int32 {
  kRealTime     = 100,
  kVeryHigh     = 150,
  kHigh         = 200,
  kStandard     = 1000,
  kLowUsage     = 1500,
  kVeryLowUsage = 2000,
};
```

1.3 III: The Team Domain System.

The team system holds process metadata for the backend scheduler to run on. It holds methods and fields for backend specific operations. One implementation of such team is the UserProcessTeam object inside NeKernel.

1.4 Illustration: Team System.

The following sample is used to hold team metadata. This is part of the NeKernel source tree.

```
class UserProcessTeam final {
 public:
  explicit UserProcessTeam();
  ~UserProcessTeam() = default;

  NE_COPY_DEFAULT(UserProcessTeam)

  Array<UserProcess, kSchedProcessLimitPerTeam>& AsArray();
  Ref<UserProcess>&                              AsRef();
  ProcessID&                                      Id() noexcept;

 public:
  UserProcessArray mProcessList;
  UserProcessRef   mCurrentProcess;
  ProcessID          mTeamId{0};
  ProcessID          mProcessCur{0};
};
```

1.5 IV: The Process Image System

The 'ProcessImage' container is a system designed to contain process data, and metadata, its purpose comes from the need to separate data and code. Such usage is useful for validation and vettability. This approach helps separate concerns and give modularity to the system, as the image and process structure are not mixed together.

1.6 Illustration: Process Image System.

The following sample is a container used to hold process data and metadata. This is part of the NeKernel source tree.

```
using ImagePtr = VoidPtr;

struct ProcessImage final {
  explicit ProcessImage() = default;

 private:
  friend UserProcess;
  friend KernelTask;

  friend class UserProcessScheduler;

  ImagePtr fCode;
  ImagePtr fBlob;

 public:
  Bool HasCode() const { return this->fCode != nullptr; }

  Bool HasImage() const { return this->fBlob != nullptr; }

  ErrorOr<ImagePtr> LeakImage() {
    if (this->fCode) {
      return ErrorOr<ImagePtr>{this->fCode};
    }
  }

  ErrorOr<ImagePtr> LeakBlob() {
    if (this->fBlob) {
      return ErrorOr<ImagePtr>{this->fBlob};
    }
  }
};
```

2 V: Conclusion.

The CPS is a system with provable domains and separation of computation, it governs how a Process Domain shall compute its child processes, it does not provide the algorithms. Which is why, one scheduler backend (such as the UserProcessScheduler) takes care of user process scheduling and fairness.

3 References

1. **CoreProcessScheduler.h (2025), github.com**
2. **NeKernel.org (2025), nekernel.org**
3. **Scheduling: Introduction (2012), pages.cs.wisc.edu**
4. **Processor Affinity, Multiple CPU Scheduling (2003), tmurgent.com**