Processes

Creating and getting information about processes.
Overview

- Processes
- Pipes
- Inter-Process Synchronization
- Shared Memory
The Process Class

- POCO provides the `Poco::Process` class that allows you to:
  - get some information about the current process
  - start a new process
  - terminate another process

- `#include "Poco/Process.h"

- All methods of `Poco::Process` are static.
Getting Information About a Process

- **Process::PID Process::id()** returns the process ID of the current thread. `Process::ID` is a platform dependent integer type.

- **void Process::times(long& userTime, long& kernelTime)** returns the number of seconds the current process has spent executing in user mode, and kernel mode, respectively.
Creating a Process

- `ProcessHandle Process::launch(const std::string& path, const std::vector<std::string>& args)` creates a new process by launching the executable specified by `path` and passing it the command line arguments given in `args`.

- `Poco::ProcessHandle` has two member functions:
  - `Process::PID ProcessHandle::id() const` returns the process ID of the newly created process.
  - `int wait() const` waits for the process to terminate and returns the exit code of the process.
Creating a Process With I/O Redirection

> ProcessHandle Process::launch(const std::string& path, const std::vector<std::string>& args, Pipe* inPipe, Pipe* outPipe, Pipe* errPipe)

creates a new process by launching the executable specified by `path` and passing it the command line arguments given in `args`.

> Pointers to `Poco::Pipe` objects for the new process' standard input, standard output and standard error channel can be passed. If a non-null pointer is passed, the corresponding channel will be redirected to the pipe.

> The same `Pipe` instance can be used for `outPipe` and `errPipe`. 
Working with Pipes

- You usually do not work with **Pipe** objects directly. Although you'll have to create instances of Pipe, for writing and reading data from a pipe you use the `Poco::PipeOutputStream` and `Poco::PipeInputStream` classes.

- #include "Poco/PipeStream.h"

- A **Pipe** is a unidirectional (half-duplex) communication channel, which means that data only flows in one direction.

- You can either read from a **Pipe**, or write to a **Pipe**, but not both with one instance.
#include "Poco/Process.h"
#include "Poco/PipeStream.h"
#include "Poco/StreamCopier.h"
#include <fstream>

using Poco::Process;
using Poco::ProcessHandle;

int main(int argc, char** argv)
{
    std::string cmd("/bin/ps");
    std::vector<std::string> args;
    args.push_back("-ax");

    Poco::Pipe outPipe;
    ProcessHandle ph = Process::launch(cmd, args, 0, &outPipe, 0);
    Poco::PipeInputStream istr(outPipe);

    std::ofstream ostr("processes.txt");
    Poco::StreamCopier::copyStream(istr, ostr);

    return 0;
}
POCO provides two primitives for inter process synchronization:

- `Poco::NamedMutex (#include "Poco/NamedMutex.h")`
- `Poco::NamedEvent (#include "Poco/NamedEvent.h")`

Both are similar to the thread synchronization primitives `Poco::Mutex` and `Poco::Event`.

Both have a name, which is used to refer to the same operating system managed mutex or event object from different processes. The name must be passed to the constructor.
NamedMutex Operations

> Poco::NamedMutex supports the same operations as Poco::Mutex:

> void NamedMutex::lock()

> bool NamedMutex::tryLock()

> void NamedMutex::unlock()

> There also is a NamedMutex::ScopedLock available.
Poco::NamedEvent only supports the following operations:

- `void NamedEvent::set()`
- `void NamedEvent::wait()`
Semantics

- **Poco::NamedMutex** and **Poco::NamedEvent** are merely references to synchronization primitives managed by the operating system.

- This differs from the thread synchronization primitives:
  - There can never be two separate **Poco::Mutex** instances that refer to the same operating system mutex object.
  - However, there can be multiple **Poco::NamedMutex** objects referencing the same operating system mutex object. Otherwise, inter thread synchronization would not be possible.
Shared Memory

- Shared Memory support in POCO is implemented by the Poco::SharedMemory class.
- #include "Poco/SharedMemory.h"
- A shared memory region can be created in two ways:
  - a named memory region of a certain size can be created
  - a file can be mapped into a shared memory region
The SharedMemory Class

- The `begin()` and `end()` member functions return a pointer to the begin and one-past-end of the shared memory region, respectively.

- The `SharedMemory` class is implemented using the Pimpl (handle/body) idiom together with reference counting, thus `SharedMemory` objects can be assigned and copied (although nothing is copied physically).
// Map a file into memory

#include "Poco/SharedMemory.h"
#include "Poco/File.h"

using Poco::SharedMemory;
using Poco::File;

int main(int argc, char** argv)
{
    File f("MapIntoMemory.dat");
    SharedMemory mem(f, SharedMemory::AM_READ); // read-only access

    for (char* ptr = mem.begin(); ptr != mem.end(); ++ptr)
    {
        // ...
    }

    return 0;
}
// Share a memory region of 1024 bytes
#include "Poco/SharedMemory.h"

using Poco::SharedMemory;

int main(int argc, char** argv)
{
    SharedMemory mem("MySharedMemory", 1024,
                     SharedMemory::AM_READ | SharedMemory::AM_WRITE);

    for (char* ptr = mem.begin(); ptr != mem.end(); ++ptr)
    {
        *ptr = 0;
    }

    return 0;
}