Rationale

Many applications require the ability to launch another application to carry out some tasks, and to do so portably. Different operating systems provide conceptually and structurally disparate means to do so. For example posix systems use the fork/exec paradigm, while windows provides the CreateProcess api call. While windows supplies most posix api calls, fork is notably missing from the windows supplied posix subsystem.

Additionally, there are times when access to platform specific functionality is required, an example is specifying security attributes...

Purpose

Boost.Process has the following aims:

- Provide a portable means for launching an executable as a child process
- Provide a framework for defining both portable and platform specific process initializers.
- Provide a collection of ready-to-use process initializers.
- Provide a method to monitor and manage child process lifetime.
- Provide attachment to child process standard in, out, err streams
- ...

The library includes stock components for the portable specification of child process' application path, working_directory, arguments, environment and standard in, out and err streams.

Overview

At the heart of Boost.Process is a collection of concepts and a set predefined models of these concepts that collaborate to simplify the portable launching of an executable as a child process. Further, users may provide their own models of these concepts to extend the functionality in a portable and/or platform specific manner.

Each platform's files are organized into separate platform directories and namespaces, minimizing #ifdef'd code and coincidental coupling between platform specific code. This makes adding a new platform orthogonal to existing platforms.

Concepts

The fundamental building blocks of the library are the concepts of an *initializer* which serves to initialize the data structures specified by an *executor*, the *executor* uses these data structures to launch the executable as a child process, a concept of a *monitor* allows the synchronization of parent and child processes, and models of a *pipe* allow streaming between parent processes child processes and their siblings. Models of *intializer* mutually visit models of *executor*. Models of *monitor* are produced by models of *executor*. Models of a *pipe* are used by models of an *initializer* to initialize child process standard stream input and output.

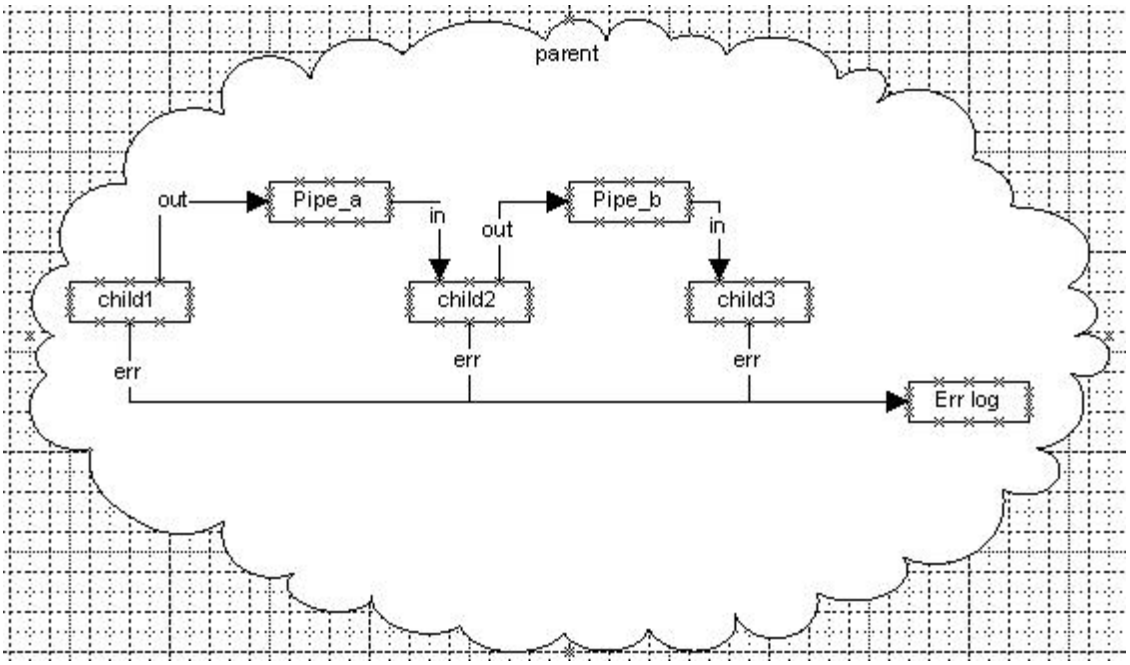
- **Initializer**
  An initializer is constructable possibly with user data which collaborates with an executor at the appropriate phases of child process creation.
  - **Windows Initializer**
    A windows initializer is a refinement of an initializer which has:
    - *pre_create* method callable before the call to the windows create process. api.
    - *post_create* method callable after the call to the windows create process. api.

        - **Posix Initializer**
          A posix initializer is a refinement of an initializer which has:
            - *pre_fork_parent* method callable before the process is forked.
            - *post_fork_parent* method callable within the parent process after the process is succesfully forked.
            - *failed_fork_parent* method callable within the parent process after the fork call fails.
            - *post_fork_child* method callable within the child process after the process is forked before the call to exec.
            - failed_exec_child method callable within the child process after the fork call fails.
        - **Portable Initializer**
          A portable initializer refines the initializer concept when there exists an initializer with the same name in each supported platform namespace.

- **Executor**
  An executor is default constructable with an execute method taking a sequence of initializers. Each initializer in the sequence is visited at each phase of child process creation by the executor, passing a reference of itself to the initializer.
    - **Windows Executor**
      A windows executor is a refinement of an executor which has an execute method taking a sequence of windows initializers. The *pre_create* method of each initializer is called before the windows process is created, and the *post_create* method of each initializer is called after the windows process is created.
    - **Posix Executor**
      A posix executor is a refinement of an executor which has an execute method taking a fusion sequence of posix initializers. The *pre_fork* method of each initializer is called before the posix process is forked, the *post_fork_parent* method of each initializer is called from the initiating parent process after fork is returned, and the *post_fork_child* method of each initializer is called from the newly forked child process only if fork succeeds.
    - **Portable Executor**
      A portable executor refines the executor concept, when there exists an initializer with the same name in each supported platform namespace.

- **Monitor**
  ...
- **Pipe**
  ...

Motivating Example



A parent process wants to create 3 child processes. Child1's standard output provides standard input to Child2 whose standard output provides standard input to Child3. The parent process opens an error log file and writes to it as should each of the Child processes. Let's implement this with boost process::

```
namespace bp = boost::process;
namespace io = boost::iostreams;

io::file_descriptor_sink log("my_error_log.txt);
```

Starting at the top. The parent process owns a log file that it's using for other things in addition to capturing it's child processs err info.

```
bp::pipe pipe_a;
bp::pipe pipe_b;
```

We first create a couple of pipes that will be used by the stdin, stdout initializers.

```
bp::monitor c1 = bp::executor().execute(make_tuple
( bp::paths("c:/some/path/exe1.exe")
, bp::arg("-d")(123)(a_boost_filesystem_path)
, bp::env_inherited()
, bp::stdin_none()
, bp::stdout_to(pipe_a)
, bp::stderr_to(log)
));
```

Next we launch child process 1 by constructing an executor, calling it's execute method, passing in a tuple(or other conforming fusion sequence) of initializers. The 1st tuple element is paths object requiring an absolute path to an executable image file. This single argument constructor also sets the working directory to the directory containing the executable. The second tuple element is an args object, which is constructed by constructing an arg object

with “-d”, and then concatenating it with additional arg objects returns an args object. bp::arg uses boost::lexical_cast<std::string> to access the string representation of the object. The 3rd tuple element makes the current environment available to the child process. The remaining tuple elements specify standard stream definitions: stdin is null, stdout writes to pipe_a, and stderr is redirecting it’s output to the parent’s log file. Finally a monitor object is initialized with the minimal process information needed to later interact with the child process.

```
bp::monitor c2 = bp::executor().execute(make_tuple
( bp::paths(“c:/some/path/exe2.exe”, boost::filesystem::current_path())
, bp::args(“--x=abcd efgh”)(a_boost_filesystem_path)(“-Z”)(123)
, bp::env_inherited()
, bp::stdin_from(pipe_a)
, bp::stdout_to(pipe_b)
, bp::stderr_to(log)
));
```

Launching Child2 demonstrates specifying the working directory equal to the parent’s current working directory. Also note Child2 will receive its input from pipe_a(generated from child1 output), and Child2 will write it’s output to pipe_b.

```
bp::monitor c3 = bp::executor().execute(make_tuple
( bp::paths(“c:/some/path/exe3.exe”, “f:/some/other/path/working_dir”)
, bp::args(“-d”)(123)(a_boost_filesystem_path)
, bp::env_none() + bp::env_vars(“var1”)(“some_value”)
, bp::stdin_from(pipe_b)
, bp::stdout_none()
, bp::stderr_to(log)
, my_portable_initializer(“on”, 123, ...)
));
```

Launching Child3 demonstrates specifying an arbitrary working directory . Note that bp::paths constructor’s arguments are boost::filesystem::path const&, here relying on implicit construction from strings. The 3rd tuple element demonstrates constructing an empty environment and adding a single var1=some_value item. The 7th tuple element demonstrates specifying a non-stock initializer.