# A proposal for a standard process management library

Emmanuel Deloget

April 4, 2013

## 1 Preamble

This document proposes an extension to the standard library.

The goal of this extension it so provide components and functions that can be used to create, manage and query processes.

### 1.1 This is a working document

This document outlines the different parts of the proposal but it's not the proposal itself. The proposal is written using LibreOffice and can be found on the Google Code repository as odt/cpp1y-process-proposal.odt (and the corresponding odt/cpp1y-process-proposal.pdf).

### 1.2 Rationale

Threads and processes are inherently similar. From an operating system point of view, they may even be implemented using the same mechanisms (as this is the case in the Linux operating system). The main difference between processes and threads lies in the way they manage virtual memory (when available): multiple threads in a process live in the the same memory space, while multiple processes on a system live in different memory space. There are advantages and disadvantages to both:

- Little-to-none security mechanism is available to protect the memory space used by a thread from another thread.
- Threads are lighter than processed. This is due to the fact that when scheduling two different processes, the operating system must manage their respective view of the memory and initialize the hardware MMU accordingly. Threads do not share this requirement as they have the same view of the memory.

This is of course a very high-level view since it does not take thread local storage into account.

Process management is an important part of system programming. An implementation would allow C++ to be used to create background programs (such as daemons in POSIX environment or services on Windows) or to start other programs or utilities (as done by a shell or a launcher UI). As of today, programmers have to use specific platform APIs to implement such functionalities in their programs, making them difficult to port to other system architectures.

### 1.3 Design consideration

Since processes and threads are quite similar, it make sense to proposes a similar interface for both classes. I decided to give this path a try and I believe it worked quite well.

## 2 Proposal

Please note that both the numbering scheme and the text below are of course subject to changes. They may not match the requirements of the text of the C++ standard. Special Development notes are added to clarify some points or to denote specific issues.

### 2.1 header <process> synopsys

This section describe components that can be used to create and manage processes. [ *Note*: these processes are intended to map one-to-one with operating system processes. — *end note* ]

```
namespace std {
 enum struct terminate_flag;

 class process;

 void swap(process& x, process& y);

 namespace this_process {
  process::id get_id() noexcept;
  template <typename Args...>
   void exec(const string& cmd, Args&&... args);
 }
}
```

### 2.2 enum struct terminate_flag

```
namespace std {
  enum struct terminate_flag
```

```
  {
    terminate_none,
    terminate_child,
    terminate_parent,
    terminate_both
  };
}
```

### 2.3 class process

The class process provides a mechanism to create a new operating system process, to join with a process (i.e., wait for a process to complete), and to perform other operations that manage and query the state of a process. A process object uniquely represents a particular operating system process. That representation may be transferred to other process objects in such a way that no two process objects simultaneously represent the same operating system process. An operating system process is detached when no process object represents that process. Objects of class process can be in a state that does not represent an operating system process. [ *Note*: A process object does not represent an operating system process after default construction, after being moved from, or after a successful call to detach or join. — *end note* ]

```
namespace std {
 class process {
 public:
  typedef implementation-defined
   native_handle_type;

  class id;

  process(process&) = delete;
  process(const process&) = delete;
  process& operator=(const process&) = delete;

  process() = default;
  process(process&& __p) noexcept;

  template <typename F, typename ...Args>
   explicit
   process(terminate_flag tflag,
    F&& f, Args&&... args);

 template <typename F, typename ...Args>
   explicit
   process(F&& f, Args&&... args);

  ~process();
```

```
    process& operator=(process&& __p) noexcept;
    void swap(process& __t) noexcept;

    void detach();
    void join();
    bool joinable() const noexcept;

    process::id get_id() const noexcept;
    native_handle_type native_handle() const noexcept;
  };
}
```

#### 2.3.1 class process:id

```
namespace std {
  class process::id
  {
  public:
    id() noexcept;
  };

  bool operator==(process::id x, process::id y);
  bool operator!=(process::id x, process::id y);
  bool operator<=(process::id x, process::id y);
  bool operator>=(process::id x, process::id y);
  bool operator<(process::id x, process::id y);
  bool operator>(process::id x, process::id y);

  template <class Char, class Traits>
    basic_ostream<Char, Traits>&
    operator<<(
      basic_ostream<Char, Traits>& out,
      process::id id);

  // hash support
  template <class T> struct hash;
  template <> struct hash<process::id>;
}
```

An object of type process::id provides a unique identifier for each process and a single distinct value for all process objects that do not represent an operating system process. Each operating system process has an associated process::id object that is not equal to the process::id object of any other operating system process and that is not equal to the process::id object of any std::process object that does not represent any operating system process.

[ *Development note*: at this point, we do not take into account some very specific implementation such as PID namespace in Linux. Under Linux, two processes can share the same process id (PID) if they do not belong to the same PID namespace. Processes in a particular PID namespaces are unaware of the existence of separate processes in a different namespace even if these processes has the same PID. — *end note* ]

process::id shall be a trivially copyable class. The library may reuse the value of a process::id of a terminated operating system process that can no longer be joined.

[ *Note*: Relational operators allow process::id objects to be used as keys in associative containers. — *end note* ]

```
id() noexcept;
```

***Effects***: construct an object of type process::id.

***Postconditions***: the object does not represent an operating system process. [ *Development note*: this doesn't mean that 0 is a good value to represent the process id. Some systems may assignate PID 0 to a particular process. — *end note* ]

```
bool operator==(process::id x, process::id y);
```

***Returns***: `true` only if x and y represent the same operating system process or neither x nor y represents any operating system process.

```
bool operator!=(process::id x, process::id y);
```

***Returns***: !(x == y)

```
bool operator<(process::id x, process::id y);
```

***Returns***: A value such that operator< is a total ordering.

```
bool operator<=(process::id x, process::id y);
```

***Returns***: (x < y) || (x == y)

```
bool operator>(process::id x, process::id y);
```

***Returns***: !(x <= y)

```
bool operator>=(process::id x, process::id y);
```

***Returns***: !(x < y)

```
 template <class Char, class Traits>
basic_ostream<Char, Traits>&
operator<<(basic_ostream<Char, Traits>& out,
process::id id);
```

***Effects*:** Inserts an unspecified text representation of id into out. For two objects of type process::id x and y, if x == y the process::id objects shall have the same text representation and if x != y the process::id objects shall have distinct text representations.

***Returns*:** out.

```
template <> struct hash<thread::id>;
```

***Requires*:** the template specialization shall meet the requirements of class template hash.

### 2.3.2 process constructors

```
process() noexcept;
```

***Effects*:** Construct an object of type process that does not represent an operating system process.

***Postcondition*:** `get_id() != id()`

```
 template <class F, class... Args>
explicit process(terminate_flag tf, F&& f, Args&&... args)
```

***Requires:*** `F` and each `Ti` in `Args` shall satisfy the `MoveConstructible` requirements. *INVOKE ( DECAY_COPY ( std::forward<F>(f) ), DECAY_COPY ( std::forward<Args> (args) ) ... )* shall be a valid expression.

***Effects*:** Construct an object of type process. The new operating system process executes *INVOKE (DECAY_COPY ( std::forward<F>(f)), DECAY_COPY ( std::forward<Args> (args) ) ...)* with the calls to *DECAY_COPY* being evaluated in the constructing process. Any return value from this invocation is ignored. [ *Note*: This implies that any exceptions not thrown from the invocation of the copy of f will be thrown in the constructing process, not the new process. — *end note* ] If the invocation of *INVOKE (DECAY_COPY ( std::forward<F>(f)), DECAY_COPY ( std::forward<Args> (args) ) ... )* terminates with an uncaught exception, `std::terminate` shall be called.
If the expression (`tf == terminate_flag::terminate_child || tf ==`

`terminate_flag::terminate_both`) evaluate to `true`, the new operating system process calls `std::exit()` after the invocation.
If the expression (`tf == terminate_flag::terminate_parent || tf == terminate_flag::terminate_both`) evaluate to `true`, the parent operating system process calls `std::exit()` if the creation of the new operating system process succeeds.

***Synchronization*:** The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of `f`.

***Postcondition*:** `get_id() == id(). *this` represents the newly started process.

***Throws*:** `system_error` if unable to start the new thread.

***Error conditions*:** [ *Development note*: not specified yet. — *end note* ]

```
template <class F, class... Args>
explicit process(F&& f, Args&&... args)
```

***Requires:*** `F` and each `Ti` in `Args` shall satisfy the `MoveConstructible` requirements. *`INVOKE ( DECAY_COPY ( std::forward<F>(f) ), DECAY_COPY ( std::forward<Args> (args) ) ... )`* shall be a valid expression.

***Effects*:** `process(terminate_flag::terminate_child, f, args...)`

```
thread(thread&& x) noexcept;
```

***Effects*:** Constructs an object of type process from x, and sets x to a default constructed state.

***Postconditions*:** x.get_id() == id() and get_id() returns the value of x.get_id() prior to the start of construction.

### 2.3.3 process destructor

```
~process();
```

If `joinable()` then `terminate()`, otherwise no effects. [ *Note*: Either implicitly detaching or joining a `joinable()` process in its destructor could result in difficult to debug correctness (for detach) or performance (for join) bugs encountered only when an exception is raised. Thus the programmer must ensure that the destructor is never executed while the process is still joinable. — *end note* ]

### 2.3.4 process assignment

`process& operator=(process&& x) noexcept;`

***Effects*:** If `joinable()`, calls `terminate()`. Otherwise, assigns the state of `x` to `*this` and sets `x` to a default constructed state.

***Postconditions*:** `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the assignment.

### 2.3.5 process members

`void swap(process& x) noexcept;`

***Effects*:** Swaps the state of `*this` and `x`.

`bool joinable() const noexcept;`

***Returns*:** `get_id() != id()`

`void join();`

***Requires*:** `joinable()` is true.

***Effects*:** Blocks until the process represented by `*this` has completed.

***Synchronization*:** The completion of the process represented by `*this` synchronizes with the corresponding successful `join()` return. [ *Note*: Operations on `*this` are not synchronized. — *end note* ]

***Postconditions*:** The process represented by *this has completed. `get_id() == id()`.

***Throws*:** `system_error` when an exception is required

***Error conditions*:** [ *Development note*: not specified yet. — *end note* ] `void detach();`

***Requires***: `joinable()` is true.

***Effects***: The process represented by `*this` continues execution without the calling process blocking. When `detach()` returns, `*this` no longer represents the possibly continuing operating system process. When the process previously represented by `*this` ends execution, the implementation shall release any owned resources.

***Postcondition***: `get_id() == id()`.

***Throws***: `system_error` when an exception is required.

***Error conditions***: [ *Development note*: not specified yet. — *end note* ]

`id get_id() const noexcept;`

***Returns***: A default constructed id object if `*this` does not represent an operating system process, otherwise `this_process::get_id()` for the operatnig system process represented by `*this`.

## 2.4 namespace this_process

```
namespace std {
 namespace this_process {
  process::id get_id() noexcept;

  template <typename Args...>
   void exec(const string& cmd, Args&&... args);
 }
}
```

`process::id this_process::get_id() noexcept;`

***Returns***: an objet of type `process::id` that uniquely identifies the current process. No other process shall have this id and this process shall always have this id. The object returned shall not compare equal to a default constructed `process::id`.

```
template <typename Args...>
void exec(const string& cmd, Args&&... args);
```

***Effects*:** replaces the current process image by a new process image. `cmd` specify the command to execute, and `args...` is the command argument list. If the underlying system call executes, the function never returns and the new process image shall have the same id as the calling process. [ *Development note*: the choice of std::string to represent the command is due to the fact that not all operating systems allows unicode strings to be used as commands. An open question on this subject is to be found below. — *end note* ]

***Throws*:** `system_error` when the underlying system call fails to execute properly.

***Error conditions*:** [ *Development note*: not specified yet. — *end note* ]

### 2.5 header <named_mutex> synopsys

```
namespace std {
  class named_mutex;
  class named_recursive_mutex;
  class named_timed_mutex;
  class named_timed_recursive_mutex;
}
```

The *named mutex types* are the following standard library types

- `std::named_mutex;`
- `std::named_recursive_mutex`;
- `std::named_timed_mutex;`
- `std::named_recursive_timed_mutex`.

They shall meet the requirements set out in this section. In this description, `nm` denotes an object of named mutex type.

The named mutex types shall meet the `Lockable` requirements.

The mutex types shall be `Destructible`. If initialization of an object of a mutex type fails, an exception of type `system_error` shall be thrown. The mutex types shall not be copyable or movable. [ *Note*: named mutex types are not `DefaultConstructible`. — *end note* ]

[ *Development note*: The error conditions for error codes, if any, reported by member functions of the named mutex types are not yet specified. — *end note* ]

The implementation shall provide lock and unlock operations, as described below. For purposes of determining the existence of a data race, these behave as atomic operations. The lock and unlock operations on a single named mutex shall appear to occur in a single total order. [ *Note:* this can be viewed as the modification order of the named mutex. — *end note* ] [ *Note:* Construction and destruction of an object of a named mutex type need not be thread-safe; other synchronization should be used to ensure that named mutex objects are initialized and visible to other processes. — *end note* ]

The expression `nm.lock()` shall be well-formed and have the following semantics:

***Requires*:** If `nm` is of type `std::named_mutex` or `std::named_timed_mutex`, the calling thread does not own the mutex.

***Effects*:** Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.

***Postcondition*:** The calling thread owns the mutex.

***Return type*:** void

***Synchronization*:** Prior `unlock()` operations on the same object shall synchronize with this operation.

***Throws*:** `system_error` when an exception is required.

***Error conditions*:** [ *Development note*: not specified yet. — *end note* ]

The expression `nm.try_lock()` shall be well-formed and have the following semantics:

***Requires*:** If `nm` is of type `std::named_mutex` or `std::named_timed_mutex`, the calling thread does not own the mutex.

***Effects*:** Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and `try_lock()` immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [ *Note*: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange. — *end note* ] An implementation should ensure that `try_lock()` does not consistently return false in the absence of contending mutex acquisitions.

***Return* type:** `bool`

***Returns*:** `true` if ownership of the mutex was obtained for the calling thread, otherwise `false`.

***Synchronization*:** If `try_lock()` returns true, prior unlock() operations on the same object synchronize with this operation. [ *Note*: Since `lock()` does not synchronize with a failed subsequent `try_lock()`, the visibility rules are weak enough that little would be known about the state after a failure, even in the absence of spurious failures. — *end note* ]

***Throws*:** Nothing.

The expression `nm.unlock()` shall be well-formed and have the following semantics:

***Requires*:** The calling thread shall own the mutex.

***Effects*:** Releases the calling thread's ownership of the mutex.

***Return type*:** `void`

***Synchronization*:** This operation synchronizes with subsequent lock operations that obtain ownership on the same object.

***Throws*:** Nothing.

## 2.6 class named_mutex

```
namespace std {
  class named_mutex
  {
  public:
    named_mutex(const std::string& name);
    ~named_mutex();

    named_mutex() = delete;
    named_mutex(const named_mutex&) = delete;
    named_mutex& operator=(const named_mutex&) = delete;

    void lock();
    bool try_lock() noexcept;
    void unlock() noexcept;

    typedef implementation-defined native_handle_type;
    native_handle_type native_handle();
```

```
    std::string name() const;
  };
}
```

The class `named_mutex` provides a non-recursive named mutex with exclusive ownership semantics. If one thread in a process owns a named mutex object, attempts by another thread in the same or in another process to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the owning thread has released ownership with a call to `unlock()`.

[ *Note*: After a thread A has called `unlock()`, releasing a named mutex, it is possible for another thread B to lock the same named mutex, observe that it is no longer in use, unlock it, and destroy it, before thread A appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread A doesn't access the named mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. — *end note* ]

The class `named_mutex` shall satisfy all the `NamedMutex` requirements.

[ *Note*: A program may deadlock if the thread that owns a named mutex object calls `lock()` on that object. If the implementation can detect the deadlock, a `resource_deadlock_would_occur` error condition may be observed. — *end note* ]

The behavior of a program is undefined if it destroys a `named_mutex` object owned by any thread in any process or a thread in any process terminates while owning a `named_mutex` object.

# 3 Technical considerations

## 3.1 About this_process::exec()

The execve() system call which is used to implement this template function has the following prototype:

```
int execve(const char *filename,
 char *const argv[],
 char *const envp[]);
```

This proposal eludes the envp parameter for now (it may appear in a subsequent proposal, see below).

The argv argument is an array of char *const. In order to derive this array from an args... list, we need a specialized mechanism. In the test implementation, I use std::stringstream (to convert any Args type into its string representation) and I allocate the individual C strings before I copy their content. The relevant (tentative) code is:

```
void __unpack_to_strings(vector<char*>& __l)
{ __l.push_back(NULL); }

template <typename _Arg0, typename... _Args>
void __unpack_to_strings(
   vector<char*>& __l,
   _Arg0&& __arg0,
   _Args&&... __args)
{
 stringstream __stream;
 __stream << __arg0;
 string __out = __stream.str();
 char *__s = new char[__out.length()+1];
 copy(__out.begin(), __out.end(), __s);
 __s[__out.length()] = 0;
 __l.push_back(__s);
 __unpack_to_strings(__l, __args...);
}
```

**Such kind of code is likely to be pervasive in C++11 codebases**: transforming an argument pack into a container of things (whatever the thing is) is very likely to be an algorithm which may be used quite often by C++ programmers. There might be room here for a standard utility function, although I understand that a generic algorithm might be difficult to devise. I may replace the code above by a better version (see below, not definitive) but ideally, a standard algorithm would be far better.

```
template <class _Adder>
void __unpack_to(_Adder&& __adder)
{ }

template <
 class _Adder,
 class _Arg0,
 class... _Args>
void __unpack_to(
 _Adder&& __adder,
 _Arg0&& __arg0,
 _Args&&... __args)
{
 __adder(__arg0);
 __unpack_to(__adder, __args...);
}
```

Another implementation would be based on a conversion from std::tuple<Args...> (and thus would first require the creation of a tuple) to a container of things - using either an implicit conversion or a user-supplied conversion.

## 3.2 Is it possible to implement fork() on a Windows system?

The Windows process management API does not propose any strict equivalent to POSIX fork(). This is outlined by Microsoft in. In the same document, Microsoft endorse CreateProcess as a rough equivalent to the fork/exec use case. While this use case is often said to be prevalent, it's not the only use case wa want to address.

However, our research found that the implementation of fork() is still feasible:

- The open source cygwin environment proposes a fork() function that works like its POSIX counterpart.
- Scilab for Windows (another open source project) has implemented its own version of fork() to help its conversion to the Windows platform.
- The Subsystem for Unix-based Applications (SUA) for Windows (also known as Interix) proposes a fully compliant POSIX subsystem on top of the Windows API, including a working fork() function.

Both implementation leverage the public, low-level NT API to create a child process that the relevant characteristics with its parent process.

## 3.3 About std::terminate_flag

The typical use of std::process is similar to the following:

```
#include <process>

int main()
{
  process p([]() {
    do_something();
    std::exit(EXIT_SUCCESS);
  });
  // continue parent code
}
```

The problem with this code is that all objets with automatic storage duration created in the lambda will not be destroyed. This may result in a resource leak (this is the case when the object creates a permanent resource such as a named shm object in Linux).

Moreoever, tests showed that the exit() call is often forgotten in that case.

The proposed solution is to introduce a flag that will hint the implementation on what to do after the child process has been created and/or executed.

### 3.4 Further implementation notes

The full interface has already been implemented for both Linux and Windows (a BSD implementation shall be quite similar to the Linux one), and the full implementation has been open-sourced on Google Code:

https://code.google.com/p/edt-process-cpp1y/

As it's a PoC implementation it may differ from this proposal by several points.

## 4 Open questions

### 4.1 Should the proposal implement environment variable management?

I have the feeling that this would be an improvement over the current proposal. Such extension would contain:

- A way to retrieve the value of an environment variable (i.e. a "getenv()" function).
- A way to setup or kill a particular environment variable (i.e. "setenv()" and "unsetenv()" functions).
- A way to list all environment variables (i.e. an "environ()" function that would fetch the content of the POSIX C variable "environ").

If such extension is implemented in this proposal then another overload of `this_process::exec()` shall be needed as well.

### 4.2 Should exec() take a basic_string<> for its cmd parameter ?

Windows _execv() function has a wide string counterpart named _wexecv(). Most posix systems don't have any equivalent to this function.

I believe that UFT8 strings can be used to represent all possible command names, but support for UTF16 command names might be of interest as well on systems that support them. It's possible to convert UTF16 to UTF8 but the conversion comes with a performance penalty.