Document number: (unassigned)
Date: 2013-04-10
Revision: 0

Emmanuel Deloget, efixo
emmanuel.deloget@efixo.com

Processes and inter-process communication

Abstract: modern operating systems provide two different ways to run tasks concurrently: thread and processes. Threads are covered by the 2011 ISO/IEC C++ standard through the means of several library and core language components. Processes are yet to be added to the language definition.
This proposal focuses on a possible definition for a process creation class tentatively named `std::process` and extends the C++ standard library with several other components that are useful to implement inter-process communication.

Table of content

Motivation and scope

Process management is of high interest to every system programmer. As soon as a project is made of more than one executable program there is good chance that some form of process management will be needed.

While full-fledged process management might be seen as a desirable goal, I believe that this is not really the case. The way processes are handled is strongly dependent on the underlying operation system. While the concepts might be common (we can search processes, list them, change their priority or kill them) the implementations vary widely – too widely perhaps to be abstracted in a common interface. The various OSes tend to expose different interfaces and when exposed interfaces are similar they often have subtle different semantics.

Thus this proposal limits itself to process creation and external program execution – by the mean of a `std::process` class and an `std::this_process::exec()` function.

Once these new components are defined we need to expose solutions to help the communication between processes. Multiple communication channels are available:

- Shared memory support allow multiple processes to share data.
- Synchronization primitives (mutexes, semaphores...) helps to reduce data or resource race.
- Stream redirection enables the creation of process pipelines.

All these communication mechanisms are collectively called IPC – for Inter-process Communication. These IPC mechanisms are abstracted in a set of coherent components which are described below.

Of course, there are other IPC mechanisms that are left untouched by this proposal, including sockets (currently worked out by the Networking subgroup), SystemV message queues, signals and so on.

Impact on the standard

This proposal does not depend on any other standard but the ISO/IEC C++11 standard.

This proposal has no impact outside the C++ standard library. In particular, no new symbol are added outside the `std` namespace. Although this may change in the future, all the proposed changes are currently contained in new library headers:

- <process>
- <named_mutex>
- <named_semaphore>
- <named_pipe>
- <shmem>
- <redirect>

(Note 1: some of these headers might be merged with existing headers in a future version

of this proposal ; for example, it might make sense to regroup the named mutex and the anonymous mutexes in the same <mutex> library header).

(Note 2: the proposal often cites the POSIX specification [POSIX] in order to detail particular choices ; this does not mean that this proposal is either based upon the POSIX specification or requires any part of this specification to be implemented.)

Design decisions

On the synopsis of class std::process

The synopsis of class **std::process** is very similar to the synopsis of class **std::thread**. This is an assumed choice that has multiple reasons.

- From a conceptual point of view, threads and processes are very similar. The only major difference between theses entities is that multiple threads in a single process shares the same memory space while multiple processes live in different memory spaces.
- Like threads, a newly created process can execute arbitrary code; one can wait for the completion of a newly created process – or it can decide that such completion is of no interest (and thus detaching the new process); any processes has an id (in the form of a process ID);

From a user point of view, it make sense to use similar mechanisms to create threads and processes.

On std::terminate_flag

Class **std::process** introduces a constructor that has no equivalent for threads. This new constructor takes an additional parameter of type **std::terminate_flag**.

```
template <class F, class... Args>
std::process::process(std::terminate_flag flag,
      F&& f, Args...&& args);
```

When a parent process is forked, the new child process is a copy of the its parent process. The execution of the child process begins right after the fork point. The typical f**ork()** code is as below:

```
void some_function_1()
{
      int result = fork();
      if (result == 0) {
            // child process
      } else if (result > 0) {
            // parent process, wait for child
            waitpid(result, NULL, 0);
      } else {
            // error !
      }
      // continued
}
```

Once the code in the child process block terminates, the execution continues afterward – meaning that both the child and the parent executes the same code. Most of the time this is not the desired behavior. The programmer is then very likely to call **exit()** at the right place, as shown below.

```
void some_function_2()
{
	int result = fork();
	if (result == 0) {
		// child process; once finished, we exit
		exit(EXIT_SUCCESS);
	} else if (result > 0) {
		// parent process
	} else {
		// error !
	}
	// continued
}
```

With the proposed std::process design – and without the help of the additional constructor, the code would look like:

```
void some_function_3()
{
	std::process([](){
		// child process; once finished, we exit
		std::exit(EXIT_FAILURE);
	}).join();
}
```

However, there is an important issue with this code : the call to the exit() function prevents the destructors of the automatic variables created in the lambda expression to execute. Such a destructor may want to release a system-wide resource that was acquired in a constructor. If the destructor is not executed then the resource will be leaked and might prevent the operating system to run as intended.

The newly added **std::process** constructor helps to prevent such situation. Depending on the value of the flag, the following behavior may be observed:

- **std::terminate_flag::terminate_child**: when the user code terminates, std::exit() is automatically called. Since this call is done outside the user-provided function object the destructors of the automatic variables that were created in the function object are called.
- **std::terminate_flag::terminate_parent**: if the fork() call succeeds, the parent process exits.
- **std::terminate_flag::terminate_both**: the child process exits after the user-provided function object has terminated and the parent exits after the fork() call has succeeded.
- **std::terminate_flag::terminate_none**: both the parent and the child processes continue their execution.

The more “classical” constructor

```
template <class F, class... Args>
std::process::process(F&& f, Args...&& args);
```

is equivalent to constructing a process with the flag value `terminate_child`. This behavior is consistent with the behavior of std::thread: when the user-provided code terminates then the process exits.

On detached processes

Detaching a thread means that the thread will not be tracked by its owning process. Despite the lack of tracking the operating system will still do the Right Thing when the thread ends its execution: thread local storage will be reclaimed and the kernel-side thread object will be destroyed. This is done automatically and follow the rule of the least surprise.

Detaching a process is a different beast.

On most modern operating systems, processes are organized in a parent-child hierarchy but being a process parent does not mean that you have full control on your child. When the parent process dies the child is transferred to another parent (on POSIX systems the *init* process (i.e. the process with a PID equal to 1) is the new parent).

On a POSIX system, a dead child process enters a state which is often called the "waitable" state. While in this state the process still consume an entry in the operating system process list – i.e. it consumes a limited operating system resource. To remove the process from this list, its parent process (which can be the *init* process) must call `waitpid()`. This is not true on Microsoft Windows where a dead child is removed from the operating system process list by the operating system itself.

As of today, POSIX does not provide any way to reparent a process. There is a way to avoid the creation of zombies in the POSIX specification by telling the system that all our children are to be detached and none are to be waited for by the parent process – this is clearly not a good idea for most programs.

Thus, some implementation will need to take additional care when detaching a process. In the mock implementation I provided (see below) I decided to use additional system resources to implement this mechanism : when a child process is detached, a detached thread is created to wait for the its death. Another solution would have been to add a `SIGCHLD` signal handler and do the wait in this handler.

The main issue here is that `waitpid()` is a blocking call which is waiting for an asynchronous event. The current C++ standard does not provide any mechanism to deal with such kind of blocking calls without resorting to threads.

Another solution might be proposed in a future version of this text in the form of a `std::detached_process` class. Such class could implement double-fork on POSIX systems. Forking twice and killing the first child will reparent the second (and now orphaned) child on the *init* process ; this is a common trick used to avoid the creation of zombie processes.

On why std::process::~process() might call std::terminate()

The arguments here are very similar to the arguments that lead to the definition of the same behavior in `std::thread::~thread()`. When a join-able process object enter its destructor it may either `detach()`, `join()` or do something else.

- Given the fact that a newly created process can communicate and synchronize with its parent process (or with other processes), automatically detaching an operating system process from its std::process object might lead to strange behaviors and bugs. This is not desirable.

- An automatic **join()** might introduce a performance bug in the user code (or even worse if the underlying operating system process is to never stop). Again, this is not a desirable solution.
- Since neither an automatic **detach()** nor an automatic **join()** are satisfactory, and since we cannot throw from a destructor, we're left with the only solution of terminating the client code.

This behavior is consistent with the behavior of std::thread, I therefore believe that it should not pose any problem.

On std::this_process::exec() and std::this_process::spawn()

The goal of std::this_process::exec() is to replace the current process image with the process image of an external program. While this is a useful tool, I got several comments during the development of this proposal that asked me to provide a simple way to start an external program in a child process.

My original proposal forced the programmer to implement this functionality by himself.

```
int main()
{
	std::process([](){
		std::this_process::exec(“/bin/ls”, “-la”, “/”);
	}).join(); // or .detach()
}
```

Such code was deemed to be simple enough to not be added in this proposal.

But then, this code cannot use any optimized fork/exec code path. Microsoft Windows proposes the WINAPI **CreateProcess()** which is an optimized fork/exec. POSIX systems provides **vfork()** as an optimized replacement for **fork()** when the next system call is **exec()** (**vfork()** omits some memory management work done by **fork()**).

In order to allow an implementation to propose such fast path, I added both the **std::this_process::spawn()** and **std::this_process::spawn_wait()** functions to this proposal.

On join() vs. wait()

While devising the interface of the std::process class I tried many different naming scheme. One of this naming scheme was to use **wait()** instead of **join()** - and **waitable()** instead of **joinable()**.

The wait/waitable terms are consistent with the process-oriented vocabulary of the POSIX specification, while the join/joinable terms are consistent with the existing **std::thread** vocabulary. I decided to go with the later for the following reasons:

- There is little to no conceptual difference between waiting for a process to terminate and joining a thread.
- There is no reason why I should impose a platform-based terminology when an similar terminology is already in use. I fully understand that the existing terminology is also platform-based (coming from the POSIX world).
- Consistency between **std::thread** and **std::process** makes the later easier to understand and less confusing.

Annotated synopsis

This chapter proposes a synopsis for the proposed addition and - when needed - explain a few specific components.

These explanations should not be seen as the technical specification of the different components, nor they are the proposed wording for this proposal.

Header <process> synopsis

```
namespace std {
	enum struct terminate_flag;

	class process;

	namespace this_process {
		inline process::id get_id();

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

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

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

}
```

`this_process::exec(cmd, args...)` replace the current process image with the image of the `cmd` program, and executes this image with the arguments `args...`. The function returns only if an error occurs (in which case a `system_error` exception is thrown). All `Ti` in `Args` shall have an existing `operator<<(ostream&, Ti)`.

Calling `this_process::spawn(cmd, args...)` is equivalent to calling (pseudo-code):

```
process([](){
	this_process::exec(cmd, args...);
}).detach();
```

Calling `this_process::spawn_wait(cmd, args..)` is equivalent to calling (pseudo-code):

```
process([](){
	this_process::exec(cmd, args...);
}).join();
```

An implementation may chose to take advantage of a particular library or system call in order to optimize this sequence.

ENUM STRUCT TERMINATE_FLAG

```
namespace std {
	enum struct terminate_flag
	{
		terminate_none,
		terminate_child,
		terminate_parent,
		terminate_both
	};
}
```

The name of this enum struct is subject to change in future version of this proposal. It bears too much resemblance with the **`std::terminate()`** symbol while, at the same time, having a completely different semantic. Keeping this name might confuse users.

Multiple other names are possible, including (but not limited to):

- completion_flag
- finish_flag
- end_flag

CLASS PROCESS

```
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;
	};
}
```

An **std::process** instance represents an operating system process.

Like threads instances, processes instances are *DefaultConstructible*, *Destructible*, *Movable* and *Swapable* but they are not *CopyConstructible* nor *Assignable*.

A process instance **p** may exist in two flavor : either it represents an existing operating system process (in which case **p.get_id() != process::id()** - such a process is said to be joinable). or it doesn't (in which case **p.get_id() == process::id()**; such a process is non-joinable).

Joining (**p.join()**) or detaching (**p.detach()**) a non-joinable process raises a system_error exception.

Joining (**p.join()**) or detaching (**p.detach()**) a joinable process makes the process instance non-joinable. Of course, **p.join()** is a blocking call.

Destroying a joinable process results in the call of **std::terminate()**. The same happens if we move a process instance into a joinable process instance.

Detaching (**p.detach()**) a process should not create an operating system resource leak.

To create a joinable process, one must construct a process instance using one of the template constructor. The arguments to these constructor must respect several requirements, including:

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

The invocation of the copied callable is done in the created process child. The DECAY_COPY is executed in the context of the parent process – any exception thrown during this copy will be thrown in the constructing process, not in the child process.

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

Calling `process(f, args...)` is equivalent to calling (via delegation or other mechanisms) `process(terminate_flag::terminate_child, f, args...)`.

CLASS PROCESS::ID

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

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

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

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

The textual representation of any `process::id` instance is not to be defined by this proposal – it ultimately depends on the vendor implementation of that class. However, it should be noted that if `x` and `y` are two instances of `process::id` then:

- `x == y` means that both `x` and `y` have the same textual representation;
- `x != y` means that `x` and `y` have different textual representation.

Of course, the specialized `hash<>` for process::id must meet the requirements of class template hash.

Additional notes, questions, answers

Microsoft Windows and fork()

I believe that this proposal can be implemented on all modern major operating system, including POSIX systems such as Linux or *BSD (including Mac OS X) and the Microsoft Windows operating system.

The WINAPI does not propose any equivalent to the fork() POSIX function yet it should be noted that implementing such a function is feasible: both Interix (a POSIX skin provided by Microsoft) and the cygwin environment provides such implementations through the means of the lower level NTAPI `ZwCreateProcess()` - which is essentially a basic `fork()`.

What should be the type of the *cmd* argument of this_process::exec() ?

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.

Should this proposal include some form of signal management?

Signals is a widely used IPC mechanism on POSIX systems. The POSIX specification proposes a complete solution to send or handle signals to other processes in the form of several system calls, including `kill()` (to send signals) or `sigaction()` (to register signal handlers). It also defines a list of 28 different signal codes, including signals that were specifically defined to communicate information to the signaled process (`SIGUSR1`, `SIGUSR2`).

However, signals are yet to be universal. Microsoft Windows does allow the possibility to install signal handlers but these handlers are restricted WRT their POSIX counterpart. Moreover, no interprocess signals are defined on the Windows platform and the operating system does not provide any way to signal a specific process (i.e. it lacks a `kill()` function).

While it's possible, I think that proposing a solution to work around these limitation is of limited interest at best: such a solution would not be interoperable with programs written in another language or with programs that were written using an earlier version of the C++ standard.

Should the proposal handle process return codes?

In the current ISO/IEC C++ standard, std::thread constructor discards the thread return code. This proposal took the same road.

However, process return codes and thread return codes don't have the same interest. Since a thread lives in the same memory space than all other threads in a particular process, it still has the possibility to communicate a pseudo return value to its parent (or to any other thread). Processes don't have this possibility.

Moreover, the return code of a process is often an important indication – one that may change the behavior of the parent process. A shell programmed in C++ should be able to get return code from its child processes as it may base the execution of further process on this result.

Factoring in process return codes is not a difficult task. It might require a change in the behavior of the **`std::process`** constructors (if the copied callable returns an int then this value should be stored) and another change here and there (**`p.join()`** might return a value, as well as **`this_process::spawn_wait()`**).

Implementation

Before I submitted this proposal I created a mock implementation of all the proposed modifications. This mock implementation is available on Google Code at the following address:

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

For those who are interested this repository also contains additional notes on the proposal itself – notes that I later decided to not add to the first version of the proposal as I do not consider them to be stable or clear enough yet.

Bibliography

[POSIX] "The Open Group Base Specifications Issue 7" ; IEEE Std 1003.1-2008. Available on the Internet at http://pubs.opengroup.org/onlinepubs/9699919799/ (might requires the creation of a free account).

Thanks

I want to thank many of my forum contacts who helped me to shape this proposal. In no particular order:

- Medinoc
- Iradrille
- Joël Lamotte
- Guillaume Beltz
- Victor Gasgas

I may have forgotten someone – in which case I hereby present my most sincere apologies.

Most if not all these fine people can be reached on the Internet at:

> http://www.developpez.net/forums/f19/c-cpp/cpp/