Paradyn Parallel Performance Tools

# ProcControlAPI Programmer's Guide

Release 9.2
June 2016

Computer Science Department
University of Wisconsin-Madison
Madison, WI 53706

Computer Science Department
University of Maryland
College Park, MD 20742
Email: dyninst-api@cs.wisc.edu

WEB: WWW.DYNINST.ORG
github.com/dyninst/dyninst

Dyn
inst

# 1. Introduction

This document describes ProcControlAPI, an API and library for controlling processes. ProcControlAPI runs as part of a *controller process* and manages one or more *target processes*. It allows the controller process to perform operations on target processes, such as writing to memory, stopping and running threads, or receiving notification when certain events occur. ProcControlAPI presents these operations through a platform-independent API and high-level abstractions. Users can describe what they want ProcControlAPI to do, and ProcControlAPI handles the details.

An example use for ProcControlAPI would be as the underlying mechanism for a debugger. A user writing a debugger could provide their own user interface and debugging strategies, while using ProcControlAPI to perform operations such as creating processes, running threads, and handling breakpoints.

ProcControlAPI exposes a C++ interface. This document assumes some familiarity with several concepts from C++, such as const types, iterators, and inheritance.

The interface for ProcControlAPI can be generally divided into two parts: an interface for managing a process (e.g., reading and writing to target process memory, stopping and running threads), and an interface for monitoring a target process for certain events (e.g., watching the target process for fork or thread creation events). The manager interface uses a set of C++ objects to represent a target process and its threads, libraries, registers and other interesting aspects. Operations performed on these C++ objects in the controller process are translated into corresponding operations on the target process. The event interface uses a callback system to notify the ProcControlAPI user of interesting events in the target process.

## 1.1. Simple Example

As an example, consider the code in Figure 1 that creates a target process and prints a message whenever that target process creates a new thread. Details on the API function used in this example can be found in latter sections of this manual, but we will provide a high level description of the operations here. Note that proper error handling and checking have been left out for brevity.

1. We start by parsing the arguments passed to the controller process, turning them into arguments that will be passed to the new target process.

```
    #include "PCProcess.h"
    #include "Event.h"
    #include <iostream>
    #include <string>

    using namespace Dyninst;
    using namespace ProcControlAPI;
    using namespace std;

4.  Process::cb_ret_t on_thread_create(Event::const_ptr ev) {
       //Callback when the target process creates a thread.
5.     EventNewThread::const_ptr new_thrd_ev = ev->getEventNewThread();
       Thread::const_ptr new_thrd = new_thrd_ev->getNewThread();

       cout << "Got a new thread with LWP " << new_thrd->getLWP() << endl;
6.     return Process::cbDefault;
    }

    int main(int argc, char *argv[]) {
       vector<string> args;

1.     //Create a new target process
       string exec = argv[1];
       for (unsigned i=1; i<argc; i++)
          args.push_back(std::string(argv[i]));
2.     Process::ptr proc = Process::createProcess(exec, args);

       //Tell ProcControlAPI about our callback function
3.     Process::registerEventCallback(EventType::ThreadCreate, on_thread_create);

       //Run the process and wait for it to terminate.
7.     proc->continueProc();
8.     while (!proc->isTerminated())
          Process::handleEvents(true);

       return 0;
     }
```

**Figure 1**

2. We ask ProcControlAPI to create a new Process using the given arguments. ProcControlAPI will spawn a new target process and leave it in a stopped state to prevent it from executing.
3. After creating the new target process we register a callback function. We ask ProcControlAPI to call our function, `on_thread_create`, when an event of type `EventType::ThreadCreate` occurs in the target process.
4. The `on_thread_create` function takes a pointer to an object of type `Event` and returns a `Process::cb_ret_t`. The Event describes the target process event that triggered this callback. In this case, it provides information about the new thread in the target process. It is worth noting that `Event::const_ptr` is a not a regular pointer, but a reference counted shared pointer. This means that we do not have to be concerned with cleaning the Event—it will be automatically cleaned when the last reference disappears. The `Process::cb_ret_t` describes what action should be taken on the process in response to this event, which is described in more detail in section 6.

5. The `Event` class has several child classes, one of which is `EventNewThread`. We start by casting the Event into an `EventNewThread` and then extract information about the new thread from the `EventNewThread`.
6. In step 6, we've finished handling the new thread event and need to tell ProcControlAPI what to do in response to this event. For example, we could choose to stop the process from further execution by returning a value of Process::cbProcStop. Instead, we choose let ProcControlAPI take its default action for an EventNewThread by returning Process::cbDefault, which is to continue the process and its new thread (which were both stopped before delivery of the callback).
7. The registering of our callback in step 3 did not actually trigger any calls to the callback function—the target process was created in a stopped state and has not yet been able to create any threads. We tell ProcControlAPI to continue the target process in this step, which allows it to execute and possibly start generating new events.
8. In this step we wait for the target process to finish executing and terminate. Calling `Process::handleEvents` blocks the controller process until an event occurs, allowing us to wait for events without needing to spin the controller process on the CPU.

# 2. Important Concepts

This section focuses on some of the more important concepts in ProcControlAPI and gives a high level overview before the detailed API is presented in Section 3.

## 2.1. Processes and Threads

There are two central classes to ProcControlAPI, `Process` and `Thread`. Each class respectively represents a single target process or thread running on the system. By performing operations on the `Process` and `Thread` objects, a ProcControlAPI user is able to control the target process and its threads.

Each `Process` is guaranteed to have at least one `Thread` associated with it. A multi-threaded process may have a `Process` object with more than one `Thread`. Each process has an address space associated with it, which can be written or read through the `Process` object. Each thread has a set of registers associated with it, which can be access through the `Thread` object.

At any one time a `Thread` will be in either a *stopped state* or a *running state*. A thread in a stopped state has had its execution paused by ProcControlAPI—the OS will not schedule the thread to run. A thread in a running state is allowed to execute as normal. A thread in a running state may block for other reasons, e.g. blocking on IO calls, but this does not affect ProcControlAPI's view of the thread state. A thread is only in the stopped state if ProcControlAPI has explicitly stopped it.

A `Process` object is not considered to have a stopped or running state—only its `Thread` objects are stopped or running. A stop operation on a `Process` triggers a stop operation on each of its `Threads`, and similarly a continue operation on a `Process` triggers continue operations on each `Thread`.

## 2.2. Callbacks

In addition to controlling a target process through the Process and Thread objects, a ProcControlAPI user can also receive notification of events that happen in that process. Examples of these events would be a new thread being created, a breakpoint being executed, or a process exiting.

The ProcControlAPI user receives notice of events through a callback system. The user can register callback function that will be called by ProcControlAPI whenever a particular type of event occurs. Details about the event are passed to the callback function via an `Event` object.

### 2.2.1. Events

Each event can be broken up into an `EventType` object and an `Event` object. The `EventType` describes a type of event that can happen, and `Event` describes a specific instance of an event happening. Each `Event` will have one and only one `EventType`.

Each `EventType` has two primary fields: its time and its code. The code field of describes what type of event occurred, e.g. `EventType::Exit` represents a target process exiting. The time field of an `EventType` represents whether the `EventType` is happening before or after will have code and will have a value of `EventType::Pre`, `EventType::Post`, or `EventType::None`.

For example, an `EventType` with time and code of `EventType::Pre` and `EventType::Exit` will occur just before a target process exits, and a code of `EventType::Exec` with a time of `EventType::Post` will occur after an exec system call occurs. In this document we will abbreviate `EventTypes` such as these as pre-exit and post-exec. Some `EventTypes` do not have a time associated with them, for example `EventType::Breakpoint` does not have an associated time and thus has a time value of `EventType::none`.

An `Event` represents an instance of an `EventType` occurring. In addition to an `EventType`, each `Event` also has pointer to the `Process` and `Thread` that it occurred on. Certain events may also have event specific information associated with them, which is represented in a sub-class of `Event`. Each EventType is associated with a specific sub-class of Event.

For example, `EventType::Library` is used to signify a shared library being loaded into the target process. When an `EventType::Library` occurs ProcControlAPI will deliver an object of type `EventLibrary`, which is a subclass of `Event`, to any registered callback functions. In addition to the information inherited from `Event`, the `EventLibrary` will contain extra information about the library that was loaded into the target process.

Table 1 shows the `Event` subclass that is used for each `EventType`. Not all `EventTypes` are available on every platform—a checkmark under the specific OS column means that the `EventType` is available on that OS.

| **EventType** | **Event Subclass** | **Linux** | **FreeBSD** | **Windows** | **BG/Q** |
|---|---|---|---|---|---|
| Stop | EventStop | ✓ | | | |
| Breakpoint | EventBreakpoint | ✓ | ✓ | ✓ | ✓ |
| Signal | EventSignal | ✓ | ✓ | ✓ | ✓ |
| UserThreadCreate | EventNewUserThread | ✓ | ✓ | | ✓ |
| LWPCreate | EventNewLWP | ✓ | | ✓ | |
| Pre-UserThreadDestroy | EventUserThreadDestroy | ✓ | ✓ | | ✓ |
| Post-UserThreadDestroy | EventUserThreadDestroy | ✓ | ✓ | | |
| Pre-LWPDestroy | EventLWPDestroy | ✓ | | ✓ | |
| Post-LWPDestroy | EventLWPDestroy | ✓ | | | |
| Pre-Fork | EventFork | | | | |
| Post-Fork | EventFork | ✓ | | | |
| Pre-Exec | EventExec | | | | |
| Post-Exec | EventExec | ✓ | ✓ | | |
| RPC | EventRPC | ✓ | ✓ | ✓ | ✓ |
| SingleStep | EventSingleStep | ✓ | ✓ | ✓ | ✓ |
| Breakpoint | EventBreakpoint | ✓ | ✓ | ✓ | ✓ |
| Library | EventLibrary | ✓ | ✓ | ✓ | ✓ |
| Pre-Exit | EventExit | ✓ | | | |
| Post-Exit | EventExit | ✓ | ✓ | ✓ | ✓ |
| Crash | EventCrash | ✓ | ✓ | ✓ | ✓ |
| ForceTerminate | EventForceTerminate | ✓ | ✓ | ✓ | |

**Table 1 – EventTypes and Events**

Details about specific events can be found in Section 3.14.

### 2.2.2. Callback Functions

Events are delivered via a callback function. A ProcControlAPI user can register callback functions for an `EventType` using the `Process::registerEventCallback` function. All callback functions must be declared using the signature:

```
Process::cb_ret_t callback_func_name(Event::ptr ev)
```

In order to prevent a class of race conditions, ProcControlAPI does not allow a callback function to perform any operation that would require another callback to be recursively delivered. At most one callback function can be running at a time.

To enforce this, the event that is passed to a callback function contains only const pointers to the triggering `Process` and `Thread` objects. Any member function that could trigger callbacks is not marked const, thus triggering a compilation error if they are called on an object passed to a callback. If the ProcControlAPI user uses const_cast or global variables to get around the const restriction it will result in a runtime error. API functions that cannot be used from a callback are mentioned in the API entries.

Operations such as `Process::stopProc`, `Process::continueProc`, `Thread::stopThread`, and `Thread::continueThread` are not safe to call from a callback function, but it would still be useful to perform these operations. ProcControlAPI allows the user to use the return value from a callback function to specify whether process or thread that triggered the event should be stopped or continued. More details on this can be found in the `Process::cb_ret_t` section of the API reference.

### 2.2.3. Callback Delivery

When ProcControlAPI needs to deliver a callback it must first gain control of a user visible thread in the controller process. This thread will be used to invoke the callback function. ProcControlAPI does not use its internal threads for delivering callbacks, as this would expose the ProcControlAPI user to race conditions.

Unfortunately, the user thread is not always accessible to ProcControlAPI when it needs to invoke a callback function. For example, the user visible thread may be performing network IO or waiting for input from a GUI when an event occurs.

ProcControlAPI uses a notification system built around the `EventNotify` class to alert the ProcControlAPI user that a callback is ready to be delivered. Once the user is notified then they can call the `Process::handleEvents` function, under which ProcControlAPI will invoke any pending callback functions.

The `EventNotify` class has two mechanisms for notifying the ProcControlAPI user that a callback is pending: writing to a file descriptor and a light-weight callback function. The `EventNotify::getFD` function returns a file descriptor that will have a byte written to it when a callback is ready. This file descriptor can be added to a `select` or `poll` to block a thread that handles ProcControlAPI events. Alternatively, the ProcControlAPI user can register a light-weight callback that is invoked when a callback is ready. This light-weight callback provides no information about the Event and may occur on another thread or from a signal handler—the ProcControlAPI user is encouraged to keep this callback minimal.

It is important for a user to respond promptly to a callback notification. A target process may remain blocked while a notification is pending. If a target process is generating many events that need callbacks, a long delay in notification could have a significant performance impact.

Once the ProcControlAPI user knows that a callback is ready to be delivered they can call `Process::handleEvents`, which will invoke all callback functions. Alternatively, if the ProcControlAPI user does not need to handle events outside of ProcControlAPI, they can continue to block in `Process::handleEvents` without going through the notification system.

## 2.3. iRPCs

An iRPC (Inferior Remote Procedure Call) is a mechanism for executing code in a target process. Despite the name, an iRPC does not necessarily have to involve a procedure call—any piece of code can be executed.

A ProcControlAPI user can invoke an iRPC by providing ProcControlAPI with a buffer of machine code and specifying a Process or Thread on which to run the machine code. ProcControlAPI will insert the machine code into the address space, save the register set, run the machine code, and then remove the machine code after execution completes. When the iRPC completes (but before the registers and memory are cleaned) ProcControlAPI will deliver an `EventIRPC` to any registered callback function. The ProcControlAPI user may use this callback to collect any results from the registers or memory used by the iRPC.

Note that ProcControlAPI will preserve the registers of the thread running the iRPC, and it will preserve the memory used by the machine code. Other memory or system state changed by the iRPC may remain visible to the target process after the iRPC completes.

The machine code for each iRPC must contain at least one trap instruction (e.g., a `0xCC` instruction on x86 family or a `0x7D821008` instruction on the PPC family). ProcControlAPI will stop executing the iRPC upon invocation of the trap. Note that the trap instruction must fall within the original machine code for the iRPC. If the iRPC calls or jumps to another piece of code that executes a trap instruction then ProcControlAPI will not treat it as the end of the iRPC.

Before an iRPC can be run it must be posted to a process or thread using the `Process::postIRPC` or `Thread::postIRPC` API functions. The `Process::postIRPC` function will select a thread to post the iRPC to. Multiple iRPCs can be posted to the same thread, but only one iRPC will run at a time—subsequent iRPCs will be queued and run after the preceding iRPC completes. If multiple iRPCs are posted to different threads in a multi-threaded process, then they may run in parallel.

An iRPC can be posted to a stopped or running thread. If posted to a stopped thread, then the iRPC will run when the thread is continued. If posted to a running thread, then the iRPC will run immediately or, if posted from a callback function, when the callback function completes.

An iRPC may be blocking or non-blocking. If a blocking iRPC is posted to any Process, then calls to `Process::handleEvents` will block until the iRPC is completed.

## 2.4. Memory Management

ProcControlAPI manages memory using a shared pointer system provided by Boost (http://www.boost.org). Many of the ProcControlAPI interface objects contain a `ptr typedef` as part of their class (e.g, `Process::ptr`). This type refers to a shared pointer that points to the object. The `const_ptr` type (e.g., `Process::const_ptr`) refers to a shared pointer that points to a constant object.

The shared pointer system will use reference counting to decide when to clean objects. The ProcControlAPI user should not explicitly clean any ProcControlAPI objects, instead they should drop their references to the objects and let them be automatically cleaned. ProcControlAPI will maintain its own references for any object that is still "live" (i.e., a process or thread that is still running) so that these objects will not be pre-maturely cleaned.

A "`NULL`" value is specified by a shared pointer using the default constructor on the `ptr` type. E.g., `Process::ptr()` represents a `NULL` pointer to a `Process`.

See the Boost web-site for more details on shared pointers.

# 3. API Reference

This section gives an API reference for all classes, functions and types in ProcControlAPI. Everything defined in this section is under the namespaces `Dyninst` and `ProcControlAPI`. These types can be accessed by prepending a `Dyninst::ProcControlAPI::` in-front of them (e.g., `Dyninst::ProcControlAPI::Process`) or by adding a using namespace directive before the references (e.g., `using namespace Dyninst; using namespace ProcControlAPI;` )

## 3.1. Process

The `Process` class is the primary handle for operating on a single target process. `Process` objects may be created by calls to the static functions `Process::createProcess` or `Process::attachProcess`, or in response to certain types of events (e.g, fork on UNIX systems).

The static functions of the `Process` class serve as a central location for performing general ProcControlAPI operations, such as `handleEvents` and `registerEventCallback` when dealing with callbacks.

**Process Declared In:**
`PCProcess.h`

**Process Types:**
`Process::ptr`
`Process::const_ptr`

The `Process::ptr` and `Process::const_ptr` respectively represent a pointer and a const pointer to a `Process` object. Both pointer types are reference counted and will cause the underlying `Process` object to be cleaned when there are no more references. ProcControlAPI will maintain internal references to any `Process` it actively controls, relinquishing those references when the process either exits or is detached.

```
enum Process::cb_action_t {
   cbDefault,
   cbThreadContinue,
   cbThreadStop,
   cbProcContinue,
   cbProcStop
}
struct Process::cb_ret_t {
   cb_ret_t(cb_action_t p) : parent(p), child(cbDefault) {}
   cb_ret_t(cb_action_t p, cb_action_t c) : parent(p), child(c)
         {}
   cb_action_t parent;
   cb_action_t child;
}
```

The `cb_ret_t` enum is used as the return type for callback functions registered through `Process::registerEventCallback()`. A callback function can specify whether the thread or process associated with its event should be stopped or continued by respectively returning cbThreadContinue, cbThreadStop, cbProcContinue, or cbProcStop. The cbDefault return value returns a Process and Thread to the original state before the event occurred.

Some events, such as process spawn or thread create involve two processes or threads. In this case the ProcControlAPI user can specify a cb_action_t value for both the parent and child using the two parameter constructor for cb_ret_t.

```
typedef Process::cb_ret_t(*cb_func_t)(Event::const_ptr)
```

The cb_func_t type is a function pointer type for functions that can handle event callbacks. The callback function gets an Event::const_ptr as input, which points to the Event that triggered the callback. The cb_func_t function should return a cb_ret_t describing what to do with the process after handling the event.

```
typedef enum {
   OSNone,
   Linux,
   FreeBSD,
   Windows
   VxWorks
   BlueGeneL
   BlueGeneP
   BlueGeneQ
} Dyninst::OSType
```

A value from this enum is returned from Process::getOS and signifies the current OS on which the target process is running.

This type is used by Process, but it is declared in the Dyninst namespace in dyntypes.h.

```
typedef std::pair<Dyninst::Address, Dyninst::Address> MemoryRegi
on;
```

The MemoryRegion type represents a region of allocated memory, and the first part of the pair is the start address, the second, the end.

**Process Static Member Functions:**

```
static Process::ptr createProcess(
   std::string executable,
   const std::vector<std::string> &argv,
   const std::vector<std::string> &envp = emptyEnv,
   const std::map<int,int> &fds = emptyFDs)
```

This function creates a new process by launching an executable file named by executable with the arguments specified by argv, the environment specified in envp, and it returns a pointer to the new Process object upon success. The new process will be created with its initial thread in the stopped state.

It is an error to call this function from a callback.

If the `fds` map is not empty, then the new process will be created with the file descriptors from the `fds' first` elements `dup2` mapped to the file descriptors in `fds' second` elements.

If `envp` is empty, the environment will be inherited from the calling process.

ProcControlAPI may deliver callbacks when this function is called.

This function returns `Process::ptr()` on error, and a subsequent call to `getLastError` returns details on the error.

```
static Process::ptr attachProcess(
    Dyninst::PID pid,
    std::string executable = "")
```

This function creates a new `Process` object by attaching to the PID specified by `pid`. The new `Process` object will be returned from this function upon success. The executable argument is optional, and can be used to assist ProcControlAPI in finding the process' executable on operating systems where this cannot be easily determined (currently on AIX). The new process will be returned with all of its threads in the stopped state.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

This function return `Process::ptr()` on error, and a subsequent call to `getLastError` returns details on the error.

```
static bool handleEvents(bool block)
```

This function causes ProcControlAPI to handle any pending debug events and deliver callbacks. When an event requires a callback ProcControlAPI needs control of the main thread in order to deliver the callback. This function gives control of the main thread to ProcControlAPI for callback delivery. A user can know when to call `handleEvents` by using the `EventNotify` interface; See Sections 2.2.3 and 0 for more details on EventNotify.

If the `block` parameter is true, then `handleEvents` will block until at least one debug event has been handled. If `block` is false then `handleEvents` returns immediately if no events are ready to be handled.

This function returns true if it handled at least one event and false otherwise.

It is an error to call this function from a callback.

```
static bool registerEventCallback(
    EventType evt,
    cb_func_t cbfunc)
```

This function registers a new callback function with ProcControlAPI. Upon receiving an event with type `evt`, ProcControlAPI will deliver a callback with that event to the `cbfunc` function. Multiple functions can be registered to receive callbacks for a single `EventType`, and a single function can be registered with multiple `EventTypes`.

If multiple callback functions are registered with a single `EventType`, then it is undefined what order those callback functions will be invoked in. In this case the `cb_ret_t` result of the last callback function called will be used to determine what stop or continue operations should be performed on the process. If a single callback function is registered for the same `EventType` multiple times, then ProcControlAPI will only invoke one call to the callback function for each instance of the `EventType`.

This function returns true on success and false on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
static bool removeEventCallback(
    EventType evt,
    cb_func_t cbfunc)
```

This function un-registers a callback that was registered with `registerEventCallback`. After a successful call to this function the callback function `cbfunc` will stop being called for events with `EventType evt`. Other callback functions registered for `evt` will not be affected. Other instances of `cbfunc` registered for different `EventTypes` will not be affected.

This function returns true if a callback was successfully removed and false otherwise. Upon an error a subsequent call to `getLastError` returns details on the error.

```
static bool removeEventCallback(EventType evt)
```

This function unregisters all callback functions associated with the `EventType evt`. After a successful call to this function ProcControlAPI will stop delivering callbacks for evt until a new callback function is registered.

This function returns true if a callback was successfully removed and false otherwise. Upon an error a subsequent call to `getLastError` returns details on the error.

```
static bool removeEventCallback(cb_func_t func)
```

This function unregisters all instances of callback function `func` from any callback with any `EventType`.

This function returns true if a callback was successfully removed and false otherwise. Upon an error a subsequent call to `getLastError` returns details on the error.

**Process Member Functions:**

```
Dyninst::PID getPid() const
```

This function returns an OS handle referencing the process. On UNIX systems this is the pid of the process.

```
Dyninst::Architecture getArchitecture() const
```

This function returns an enum that describes the architecture of the target process. See Appendix A for the definition of `Dyninst::Architecture`.

```
Dyninst::OSType getOS () const
```

This function returns an enum that describes the OS of the target process. See the beginning of this section for the definition of `Dyninst::OSType`.

```
bool supportsLWPEvents () const
```

This function returns true if the target process can throw LWP create and destroy events and false otherwise.

`bool supportsUserThreadEvents () const`

This function returns true if the target process can throw user thread create and destroy events and false otherwise.

`bool supportsFork () const`

This function returns true if the fork system call is supported in the target process and false otherwise.

`bool supportsExec () const`

This function returns true if the exec system call is supported in the target process and false otherwise.

`bool isTerminated() const`

This function returns true if the target process has terminated (either via a crash or normal exit) or if the ProcControlAPI has detached from the target process. It returns false otherwise.

`bool isExited() const`

This function returns true of the target process exited via a normal exit (e.g, calling the `exit` function or returning from `main`). It returns false otherwise.

`int getExitCode() const`

If a target process exited normally then this function returns its exit code. The return result of this function is undefined if the `Process' isExited` function returns `false`.

`bool isCrashed() const`

This function returns true if the target process exited because of a crash. It returns false otherwise.

`int getCrashSignal() const`

If a target process exited because of a crash, then this function returns the signal that caused the target process to crash. The return result of this function is undefined if the `Process' isCrashed` function returns `false`.

`bool hasStoppedThread() const`

This function returns true if the target process has at least one thread in the stopped state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool hasRunningThread() const`

This function returns true if the target process has at least one thread in the running state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool allThreadsStopped() const`

This function returns true if all threads in the target process are in the stopped state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool allThreadsRunning() const`

This function returns true if all threads in the target process are in the running state. It returns false otherwise or if an error occurs. In the event of an error a call to `getLastError` returns details on the error.

`bool allThreadsRunningWhenAttached() const`

This function returns true if all threads were running when the controller process attached to this process. It returns false if any threads were stopped. If the target process was created instead of attached, this function returns true.

`bool continueProc()`

This function will move all threads in the target process into the running state. This function returns `true` if at least one thread was continued as part of the call, and `false` otherwise.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

This function return `false` on error, and a subsequent call to `getLastError` returns details on the error.

`bool stopProc()`

This function will move all threads in the target process into the stopped state. This function returns `true` if at least one thread was stopped as part of the call, and `false` otherwise.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

This function return `false` on error, and a subsequent call to `getLastError` returns details on the error.

`bool detach(bool leaveStopped = false)`

This function will detach ProcControlAPI from the target process. ProcControlAPI will no longer be able to control or receive events from the target process. All breakpoints will be removed from the target. This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

If the `leaveStopped` parameter is set to `true`, and the process is in a stopped state, then the target process will be left in a stopped state after the detach.

It is an error to call this function from a callback.

`bool temporaryDetach()`

This function temporarily detaches from the target process, but leaves the Process data structure intact. This functionality is commonly called detach-on-the-fly. The target process will not report new events nor be controllable or able to be queried by the user. Breakpoints are removed from the process. The `reAttach` function will reconnect the process after this call.

This function returns true on success and false upon error.

It is an error to call this function from a callback.

`bool reAttach()`

This function reconnects to the target process after a `temporaryDetach` call. Any breakpoints will be re-inserted back into the function, and if threads have been created or destroyed during the time detached new events will be thrown for them.

This function returns true on success and false upon error.

It is an error to call this function from a callback.

`bool terminate()`

This function forcefully terminated the target process. Upon a successful call to this function the target process will end execution. The `Process` object will record the target process as having crashed. This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

It is an error to call this function from a callback.

```
const ThreadPool &threads() const
ThreadPool &threads()
```

These functions respectively return a const reference or a reference to the Process' `ThreadPool`. The `ThreadPool` object can be used to iterate over and query the `Process' Thread` objects—see the Section 3.6 for more details on `ThreadPool`.

```
const LibraryPool &libraries() const
LibraryPool &libraries()
```

These functions respectively return a const reference or a reference to the Process' `LibraryPool`. The `LibraryPool` object can be used to iterate over and query the `Process' Library` objects—see the Section 3.7 for more details on `LibraryPool`.

`bool addLibrary(std::string libname)`

This function causes the specified library to be loaded into the process. It will trigger an event (and thus a user callback) for each library loaded (including dependencies).

```
void *getData () const
void setData (void *p) const
```

These functions respectively get and set an opaque data object that can be associated with this process. The data is not interpreted by ProcControlAPI, but remains associated with the process.

`unsigned getMemoryPageSize() const`

This function returns memory page size for the current OS on which the target process is running.

```
Dyninst::Address mallocMemory(size_t long size)
Dyninst::Address mallocMemory(
     size_t size,
     Dyninst::Address addr)
```

These functions allocate a region of memory in the target process' address space of size `size`. Upon a successful call these functions will map an area of memory in the target process that is readable, writeable and executable. The `mallocMemory(size_t)` function will allocate memory at any available address. The `mallocMemory(size_t,` Dyninst::Address) function will only allocate memory at the specified address, addr.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

Upon success these functions return the start address of memory that was allocated and 0 otherwise. Upon an error a subsequent call to getLastError returns details on the error.

```
bool freeMemory(Dyninst::Address addr)
```

This function will free a region of memory that was allocated by the mallocMemory function. Upon a successful call to this function, the area of memory starting at addr will be unmapped and no longer accessible to the target process. It is an error to call this function with an address that was not returned by mallocMemory.

It is an error to call this function from a callback.

ProcControlAPI may deliver callbacks when this function is called.

Upon success this function returns true, otherwise it returns false. Upon an error a subsequent call to getLastError returns details on the error.

```
bool writeMemory(
     Dyninst::Address addr,
     void *buffer,
     size_t size) const
```

This function writes to the target process's memory. The addr parameter specifies an address in the target process to which ProcControlAPI should write. The buffer and size parameters specify a region of controller process memory that will be copied into the target process.

It is an error to call this function on a Process that does not have at least one Thread in a stopped state.

This function returns true on success and false on error. Upon an error a subsequent call to getLastError returns details on the error.

```
bool readMemory(
     void *buffer,
     Dyninst::Address addr,
     size_t size) const
```

This function reads from the target process' memory. The addr and size parameters specify an address in the target process from which ProcControlAPI should read. The buffer parameter specifies an address in the controller process where ProcControlAPI should write the copied bytes.

It is an error to call this function on a Process that does not have at least one Thread in a stopped state.

This function returns true on success and false on error. Upon an error a subsequent call to getLastError returns details on the error.

```
bool getMemoryAccessRights(
      Dyninst::Address addr,
      mem_perm& rights)
bool setMemoryAccessRights(
      Dyninst::Address addr,
      size_t size,
      mem_perm rights,
      mem_perm& oldRights)
```

These functions respectively get and set memory permission at the specified address, `addr`. The `setMemoryAccessRights` function also affects a region of memory in the target process's address space of size `size`.

```
bool findAllocatedRegionAround(
      Dyninst::Address addr,
      MemoryRegion& memRegion)
```

This function finds a region of allocated memory `memRegion` contains address `addr`, and returns `true` on success, otherwise `false`.

```
bool addBreakpoint(
      Dyninst::Address addr,
      Breakpoint::ptr bp) const
```

This function inserts the `Breakpoint` specified by `bp` into the target process at address `addr`. See the Section 3.4 for more details on `Breakpoint`.

It is an error to call this function on a `Process` that does not have at least one `Thread` in a `stopped` state.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool rmBreakpoint(
      Dyninst::Address addr,
      Breakpoint::ptr bp) const
```

This function removes the `Breakpoint` specified by `bp` at address `addr` from the target process. See the section 3.4 on `Breakpoint` for more details.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool postIRPC(IRPC::ptr irpc) const
```

This function posts the given `irpc` to the `Process`. ProcControlAPI selects a `Thread` from the `Process` to run the iRPC and put `irpc` into that `Thread`'s queue of posted IRPCs. See Sections 2.3 and 3.5 for more information on iRPCs.

Each instance of an `IRPC` object can be posted at most once. It is an error to attempt to post a single `IRPC` object multiple times.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool runIRPCSync(IRPC::ptr irpc)`

This function posts an irpc, similar to `Process::postIRPC`; continues the thread the `irpc` was posted to; and returns when the `irpc` has completed running. The thread will be returned to its original running state when this function returns.

This function returns true if the `irpc` was successfully run, and false otherwise. Note that stopping the thread that is running the `irpc` while this function waits for `irpc` completion causes this function to return an error.

It is an error to call this function from a callback.

`bool runIRPCAsync(IRPC::ptr irpc)`

This function posts an `irpc`, similar to `Process::postIRPC`, and then continues the thread the `irpc` was posted to.

This function returns true if the `irpc` was successfully posted and run, and false otherwise.

It is an error to call this function from a callback.

`bool getPostedIRPCs(std::vector<IRPC::ptr> &rpcs) const`

This function returns all `IRPCs` posted to this `Process` in the `rpcs` vector. This list does not include any `IRPCs` currently running—see `Thread::getRunningIRPC`() for this functionality.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

### 3.1.1. mem_perm

The `mem_perm` nested class, which defined within `Process` class, represents general memory page permission for the given memory page in the process.

**mem_perm Declared In:**

`PCProcess.h`

**mem_perm Types:**

```
Process::mem_perm::read
Process::mem_perm::write
Process::mem_perm::execute
```

The Process::mem_perm::read, Process::mem_perm::write, and Process::mem_perm::execute, just as their names imply, respectively represent read, write, and execution permission of given memory page.

**mem_perm Member Functions:**

```
mem_perm() : read(false), write(false), execute(false) {}
mem_perm(const mem_perm& p) : read(p.read), write(p.write), execute(p.execute) {}
mem_perm(bool r, bool w, bool x) : read(r), write(w), execute(x) {}
```

These constructors provide a convenient way to create the specific memory permission for the given page.

```
bool getR() const
bool getW() const
bool getX() const
```

These functions return true if the given memory page has read, write, and execution permission, respectively, and false otherwise.

```
bool isNone() const
bool isR()    const
bool isX()    const
bool isRW()   const
bool isRX()   const
bool isRWX()  const
```

These functions return true if the permission of given memory page is NO_ACCESS, READ_ONLY, EXECUTE, READ_WRITE, READ_EXECUTE, and READ_WRITE_EXECUTE, respectively, and false otherwise.

```
Process::mem_perm& setR()
Process::mem_perm& setW()
Process::mem_perm& setX()
```

These functions enable read, write, and execution permission for the given page, respectively, and return this `mem_perm`.

```
Process::mem_perm& clrR()
Process::mem_perm& clrW()
Process::mem_perm& clrX()
```

These functions disable read, write, and execution permission for the given page, respectively, and return this `mem_perm`.

```
bool operator==(const mem_perm& p) const
```

This function returns true if memory permission `p` is the same as this `mem_perm` and false otherwise.

```
bool operator!=(const mem_perm& p) const
```

This function returns true if memory permission `p` is different from this `mem_perm` and false otherwise.

```
bool operator<(const mem_perm& p) const
```

This function returns true if this `mem_perm` is less than `p` according to the notation that read permission encodes to 4, write, 2, and execute, 1, and false otherwise.

```
std::string getPermName()
```

Return the memory permission name for this `mem_perm`.

## 3.2. Thread

The `Thread` class represents a single thread of execution in the target process. Any `Process` has at least one `Thread`, and multi-threaded target processes may have more. Each `Thread` has an associated integral value known as its LWP, which serves as a handle for communicating with the OS about the thread (e.g., a PID value on Linux). On some systems, depending on availability, a `Thread` may have information from the user space threading library.

**Thread Declared In:**
`PCProcess.h`

**Thread Types:**
`Thread::ptr`
`Thread::const_ptr`

The `Thread::ptr` and `Thread::const_ptr` respectively represent a pointer and a const pointer to a `Thread` object. Both pointer types are reference counted and cause the underlying `Thread` object to be cleaned when there are no more references. ProcControlAPI maintains internal references to any `Thread` it actively controls, relinquishing those references when the thread exits or is detached.

**Thread Member Functions:**

`Dyninst::LWP getLWP() const`

This function returns an OS handle for this thread. On Linux this returns a `pid_t` for this thread. On FreeBSD, this returns a lwpid_t.

`Process::ptr getProcess()`
`Process::const_ptr getProcess() const`

These functions return a pointer to the `Process` object that contains this thread.

`bool isStopped() const`

This function returns true if this thread is in a stopped state and false otherwise.

`bool isRunning() const`

This function returns true if this thread is in a running state and false otherwise.

`bool isLive() const`

This function returns true if this thread is alive, and it returns false if this thread has been destroyed.

`bool isDetached() const`

This function returns true if this thread has been detached via `Process::temporaryDetach` and false otherwise.

`bool isInitialThread() const`

This function returns true if this thread is the initial thread for the process and false otherwise.

`bool stopThread()`

This function moves the thread to into a stopped state. Upon a successful call to this function the `Thread` object will be paused and will not resume execution until the `Thread` is continued. It is an error to call this function from a callback. Instead of calling this function, a callback can stop a thread by returning `Process::cbThreadStop` or `Process::cbProcStop`.

ProcControlAPI may deliver callbacks when this function is called.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool continueThread()`

This function moves the thread into a running state. It is an error to call this function from a callback. Instead of calling this function, a callback can stop a thread by returning `Process::cbThreadContinue` or `Process::cbProcContinue`.

ProcControlAPI may deliver callbacks when this function is called.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool getRegister(
      Dyninst::MachRegister reg,
      Dyninst::MachRegisterVal &val) const
```

This function gets the value of a single register from this thread. The register is specified by the `reg` parameter, and the value of the register is returned by the `val` parameter. See Appendix A for an explanation of the `MachRegister` class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool getAllRegisters(RegisterPool pool) const`

This function reads the values of every register in the thread and returns them as part of the `RegisterPool` object `pool`. Depending on the OS, this call may be more efficient that calling `Thread::getRegister` multiple times. See Section 3.8 for a discussion of the RegisterPool class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool setRegister(
      Dyninst::MachRegister reg,
      Dyninst::MachRegisterVal val) const
```

This function writes the value of a single register in this thread. The register is specified by the `reg` parameter, and the value that should be written is specified by the `val` parameter. See Appendix A for an explanation of the `MachRegister` class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool setAllRegisters(RegisterPool &pool) const`

This function sets the values of every register in this thread to the values specified in the `RegisterPool` object `pool`. Depending on the OS, this call may be more efficient that calling `Thread::setRegister` multiple times. See Section 3.8 for a discussion of the `RegisterPool` class.

It is an error to call this function on a thread that is not in the stopped state.

Upon success this function returns `true`, otherwise it returns `false`. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool haveUserThreadInfo() const;`

This function returns true if information about this `Thread's` underlying user-level thread is available.

`Dyninst::THR_ID getTID() const;`

This function returns the unique identifier for the user-level thread. This value is only valid if `haveUserThreadInfo` returns true.

`Dyninst::Address getStartFunction() const;`

This function returns the address of the initial function for the user-level thread. This value is only valid if `haveUserThreadInfo` returns true.

`Dyninst::Address getStackBase() const;`

This function returns the address of the bottom of the user-level thread's stack. This value is only valid if `haveUserThreadInfo` returns true.

`unsigned long getStackSize() const;`

This function returns the size in bytes of the user-level thread's allocated stack. This value is only valid if `haveUserThreadInfo` returns true.

`Dyninst::Address getTLS() const;`

This function returns the address of the user-level thread's thread local storage area. This value is only valid if `haveUserThreadInfo` returns true.

```
bool readThreadLocalMemory(
     void *buffer,
     Library::const_ptr lib,
     Dyninst::Offset tls_symbol_offset,
     size_t size) const
```

This function reads from a symbol in thread local storage (TLS) memory. TLS is memory that is local to a thread and has a lifetime matching the thread. The `tls_symbol_offset` is the TLS symbol's offset in `lib`, and can be found by reading a TLS symbol's value. The `lib` parameter can point to a library or the executable. The `buffer` parameter specifies an address in the controller process where ProcControlAPI should write the copied bytes.

It is an error to call this function on a `Thread` that is not in a `stopped` state. It is also an error to call this function on a `Thread` that has not have user-level thread information, which can be tested with `haveUserThreadInfo`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool writeThreadLocalMemory(
      Library::const_ptr lib,
      Dyninst::Offset tls_symbol_offset,
      const void *buffer,
      size_t size) const
```

This function writes to a symbol in thread local storage (TLS) memory. TLS is memory that is local to a thread and has a lifetime matching the thread. The `tls_symbol_offset` is the TLS symbol's offset in `lib`, and can be found by reading a TLS symbol's value. The `lib` parameter can point to a library or the executable. The `buffer` parameter specifies an address in the controller process where ProcControlAPI should read the bytes to be copied.

It is an error to call this function on a `Thread` that is not in a `stopped` state. It is also an error to call this function on a `Thread` that has not have user-level thread information, which can be tested with `haveUserThreadInfo`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool getThreadLocalAddress(
      Library::const_ptr lib,
      Dyninst::Offset tls_symbol_offset,
      Dyninst::Address &result_addr) const
```

This function looks up the address of a symbol in thread local storage (TLS) memory. The `tls_symbol_offset` is the TLS symbol's offset in `lib`, and can be found by reading a TLS symbol's value. The `lib` parameter can point to a library or the executable. The `result_addr` parameter will be set to the target address for the TLS symbol in this Thread.

It is an error to call this function on a `Thread` that is not in a `stopped` state. It is also an error to call this function on a `Thread` that has not have user-level thread information, which can be tested with `haveUserThreadInfo`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

```
bool postIRPC(IRPC::ptr irpc) const
```

This function posts the given `irpc` to the `Thread`. The `IRPC` is put irpc into the `Thread's` queue of posted `IRPCs` and will be run when ready. See Section 2.3 for more information on posting `IRPCs`.

Each instance of an `IRPC` object can be posted at most once. It is an error to attempt to post a single `IRPC` object multiple times.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`bool getPostedIRPCs(std::vector<IRPC::ptr> &rpcs) const`

This function returns all `IRPCs` posted to this `Thread` in the vector `rpcs`. This does not include any running `IRPC`.

This function returns `true` on success and `false` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`IRPC::const_ptr getRunningIRPC() const`

This function returns a const pointer to any `IRPC` that is actively running on this `Thread`. If there is no `IRPC` actively running, then this function returns `IRPC::const_ptr()`.

`void setSingleStepMode(bool mode) const`

This function sets whether a `Thread` is in single-step mode. If called with a `mode` of `true`, then the `Thread` is put in single-step mode. If called with a `mode` of `false`, then the `Thread` is taken out of single-step mode.

A `Thread` in single-step mode will pause execution at each instruction and trigger an `EventSingleStep` event. After each `EventSingleStep` is handled (and presuming the `Thread` is still running and in single-step mode) the `Thread` will execute one more instruction and trigger another `EventSingleStep`.

`bool getSingleStepMode() const`

This function returns `true` if the `Thread` is in single-step mode and `false` otherwise.

`void *getData() const`
`void setData(void *p) const`

These functions respectively get and set an opaque data object that can be associated with this `Thread`. The data is not interpreted by ProcControlAPI, but remains associated with the `Thread`.

## 3.3. Library

A `Library` represents a single shared library (frequently referred to as a DLL or DSO, depending on the OS) that has been loaded into the target process. In addition, a `Library` will be used to represent the process' executable. `Process`' with statically linked executables will only contain the single `Library` that represents the executable.

Each Library contains a *load address* and a file name. The load address is the address at which the OS loaded the library, and the file name is the path to the library's file. Note that on some operating systems (Linux, Solaris, BlueGene, FreeBSD) the load address does not necessarily represent the beginning of the library in memory; instead it is a value that can be added to a library's symbol offsets to compute the dynamic address of a symbol.

Libraries may be loaded and unloaded by the process during execution. A library load or unload can trigger a callback with an EventLibrary parameter. The current list of libraries loaded into a process can be accessed via a `Process`' `LibraryPool` object (see Section 3.7).

**Library Types**

`Library::ptr`

`Library::const_ptr`

The `Library::ptr` and `Library::const_ptr` types are respective typedefs for a pointer and a const pointer to a library.

These pointers are not shared pointers—ProcControl will automatically clean a Library object when it is unloaded. It is not recommended that the user maintains copies of pointers to Library objects after an `EventLibrary` delivers notice of a library unload.

**Library Member Functions**

`std::string getName() const`

Returns the file name for this `Library`.

`std::string getAbsoluteName() const`

Returns a file name for this `Library` that does not contain symlinks or a relative path.

`Dyninst::Address getLoadAddress() const`

Returns the load address for this `Library`.

`Dyninst::Address getDataLoadAddress() const`

The AIX operating system can have two load addresses for a library: one for the code region and one for the data region. On AIX `Library::getLoadAddress` returns the load address of the code region and `Library::getDataLoadAddress` returns the load address of the data region. On non-AIX systems this function returns 0.

`Dyninst::Address getDynamicAddress() const`

On ELF based systems (FreeBSD, Linux, BlueGene) this function will return the address of the Library's dynamic section. On other systems this function returns 0.

`bool isSharedLib() const`

This function returns `true` of this `Library` object is refers to a shared library and `false` if it refers to an executable.

`void *getData() const`

This function returns an opaque data object that user code can associate with this library. Use `setData` to set this opaque value.

`void setData(void *p) const`

This function sets associates an opaque data object with the `Library`. ProcControlAPI does not try to interpret this value, but will return it with the `getData` function.

## 3.4. Breakpoint

A breakpoint is a point in the code region of a target process that, when executed, stops the process execution and notifies ProcControlAPI. Upon being continued the process will resume execution at the point. A `Breakpoint` object is a handle that can represents one or more breakpoints in one or more processes. Upon receiving notification that a breakpoint has executed, ProcControlAPI will deliver a callback with an `EventBreakpoint`, (see Section 3.15.7).

Some `Breakpoint` objects can be created as *control-transfer breakpoints*. When a process is continued after executing a control-transfer the process will resume at an alternate location, rather than at the breakpoint's installation point.

A single `Breakpoint` can be inserted into multiple locations within a target process. This can be useful when a user has wants to perform a single action at multiple locations in a target process. For example, if a user wants to insert a breakpoint at the entry to function `foo`, and `foo` has multiple instantiations in a process, then a single `Breakpoint` can be inserted at each instance of `foo`.

A single `Breakpoint` object can be inserted into multiple target processes at the same time. When a process does an operation that copies an address space, such as fork on UNIX, the child process will receive all `Breakpoint` objects that were installed in the parent process.

Multiple `Breakpoint` objects can be inserted into the same location with-in the same process. When this location is executed in the target process a single callback will be delivered, and the `EventBreakpoint` object will contain a reference to each Breakpoint inserted at the location. At most one control-transfer breakpoint can be inserted at any one point in a process.

Due to the many-to-many nature of `Breakpoints` and `Processes`, a single installation of a `Breakpoint` can be identified by a `Breakpoint`, `Process`, `Address` triple. The functions for inserting and removing breakpoints (`Process::addBreakpoint` and `Process::rmBreakpoint`) need all three pieces of information.

A `Breakpoint` can be a hardware breakpoint or a software breakpoint. A hardware breakpoint is typically implemented by setting special debug register in the process and can trigger on code execution, data reads or data write. A software breakpoint is typically implemented by writing a special instruction into a code sequence and can only be triggered by code execution. There are typically a limited number of hardware breakpoints available at the same time.

**Breakpoint Types**

`Breakpoint::ptr`
`Breakpoint::const_ptr`

The `Breakpoint::ptr` and `Breakpoint::const_ptr` types are respectively a pointer and a const pointer to a `Breakpoint` object. These pointers are shared pointers, and the underlying `Breakpoint` object will be automatically clean when there are no more references to it. ProcControlAPI will automatically maintain at least one reference to any Breakpoint that is installed in a target process.

**Breakpoint Constant Values**

static const int BP_X = 1
static const int BP_W = 2
static const int BP_R = 4

These constant values are used to set execute, write and read bits on hardware breakpoints.

**Breakpoint Static Functions**

`Breakpoint::ptr newBreakpoint()`

This function creates a new software `Breakpoint` object and returns it. The `Breakpoint` is not inserted into a `Process` until it is passed to `Process::addBreakpoint()`.

`Breakpoint::ptr newTransferBreakpoint(Dyninst::Address ctrl_to)`

This function creates a new control transfer software breakpoint. Upon resumption after executing this Breakpoint, control will resume at the address specified by the `ctrl_to` parameter.

```
Breakpoint::ptr newHardwareBreakpoint(unsigned int mode,
    unsigned int size)
```

This function creates a new hardware breakpoint. The `mode` parameter is a bitfield that contains an OR combination the values `BP_X`, `BP_W` and `BP_R`. These control whether the breakpoint will trigger when its target address is executed, written or read.

The `size` parameter specifies a range of memory that this breakpoint monitors. If memory is accessed between the target address and target address + `size`, then the breakpoint will trigger.

**Breakpoint Member Functions**

`bool isCtrlTransfer() const`

This function returns `true` if the `Breakpoint` is a control transfer breakpoint, and `false` if it is a regular `Breakpoint`.

`Dyninst::Address getToAddress() const`

If this `Breakpoint` is a control transfer breakpoint, then this function returns the address to which it transfers control. If this `Breakpoint` is not a control transfer breakpoint, then this function returns 0.

`void setData(void *data) const`

This function sets the value of an opaque handle that is associated with each `Breakpoint`. The opaque handle can be any value, and it can be retrieved with the `getData` function.

`void *getData() const`

This function returns the value of the opaque handled that is associated with this `Breakpoint`.

`void setSuppressCallbacks(bool val)`

This function can be used to suppress callbacks stemming from a specific breakpoint when called with `val` set to `true` value. All other effects from this breakpoint will still occur, but it will not generate a callback. By default callbacks occur from every breakpoint.

`bool suppressCallbacks() const`

This function returns `true` if callbacks have been suppressed for this breakpoint from `Breakpoint::setSuppressCallbacks` and false otherwise.

## 3.5. IRPC

`IRPC` is a class representing an Inferior Remote Procedure Call that can be run in a target process. See Section 2.3 for a high level discussion of iRPCs. Also see `Process::postIRPC` and `Thread::postIRPC` for information about posting an `IRPC`.

**IRPC Declared In:**
`PCProcess.h`

**IRPC Types:**
`IRPC::ptr`
`IRPC::const_ptr`

The `IRPC::ptr` and `IRPC::const_ptr` respectively represent a pointer and a const pointer to an `IRPC` object. Both pointer types are reference counted and will cause the underlying `IRPC` object to be cleaned when there are no more references. ProcControlAPI will maintain internal references to any `IRPC` currently posted or executing.

**IRPC Static Member Functions:**

```
IRPC::ptr createIRPC(
     void *binary_blob,
     unsigned int size,
     bool non_blocking = false)

IRPC::ptr createIRPC(
     void *binary_blob,
     unsigned int size,
     Dyninst::Address addr,
     bool non_blocking = false)
```

The `createIRPC` static function creates and returns a new IRPC object. The `binary_blob` and `size` parameters specify a buffer of machine code bytes that this IRPC should execute when invoked. ProcControlAPI will maintain its own copy of the `binary_blob` buffer, the ProcControlAPI user can free the buffer once this function completes.

If the `non_blocking` parameter is true then calls to `Process::handleEvents` will block until this IRPC is completed.

If the `addr` parameter is given, then ProcControlAPI will write and run the binary code at `addr`. Otherwise ProcControlAPI will select a location at which to run the IRPC.

**IRPC Member Functions:**

```
Dyninst::Address getAddress() const
```

The `getAddress` function returns the address at which the `IRPC` will be run. If the `IRPC` was not given an address at construction and has not yet started running, then this function may return 0.

`void *getBinaryCodeBlob() const`

The `getBinaryCodeBlob` returns a pointer to memory that contains the binary code for this IRPC.

`unsigned int getBinaryCodeSize() const`

The `getBinaryCodeSize` function returns the size of the binary code blob buffer.

`unsigned long getID() const`

The `getID` function returns an integer identifier that uniquely identifies this IRPC.

`void setStartOffset(unsigned long off)`

By default an `IRPC` will start executing its code blob at the beginning of the blob. This function can be used to tell ProcControlAPI to start execution of the code blob at some byte offset, `off`, into the blob.

This function should be called before the IRPC is posted.

`unsigned long getStartOffset() const`

If a start offset has been set for this `IRPC`, then `getStartOffset` returns it. Otherwise this function returns 0.

## 3.6. ThreadPool

A `ThreadPool` object is a collection for holding the `Threads` that make up a `Process`. Each `Process` object has one `ThreadPool` object, and each `ThreadPool` object has one or more `Thread` objects. A `ThreadPool` is typically used to iterate over or search the set of `Threads`.

Note that it is not safe to make assumptions about having consistent contents of a `ThreadPool` for a running target process. As the target process runs `Thread` objects may be inserted or removed from the `ThreadPool`. It is generally safer to stop a `Process` before operating on its `ThreadPool`. When used on a running process the `ThreadPool` iterator methods guarantee that they will not return invalid `Thread` objects (e.g, nothing that would lead to a segfault), but they do not guarantee that the `Thread` objects will refer to live threads or that they return all `Threads`.

**ThreadPool Declared In:**
`PCProcess.h`

**ThreadPool Types:**
```
class iterator {
public:
     iterator();
     ~iterator();
     Thread::ptr operator*() const;
     bool operator==(const iterator &i);
     bool operator!=(const iterator &i);
     ThreadPool::iterator operator++();
     ThreadPool::iterator operator++(int);
};

class const_iterator {
public:
     const_iterator();
     ~const_iterator();
     Thread::const_ptr operator*() const;
     bool operator==(const const_interator &i);
     bool operator!=(const const_iterator &i);
     ThreadPool::const_iterator operator++();
     ThreadPool::const_iterator operator++(int);
};
```

The `iterator` and `const_iterator` types of ThreadPool are respectively C++ iterators and const iterators over the set of threads represented by the ThreadPool. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

**ThreadPool Member Functions:**

`ThreadPool::iterator begin()`
`ThreadPool::const_iterator begin() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the beginning of the set of `Thread` objects.

`ThreadPool::iterator end()`
`ThreadPool::const_iterator end() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the iterator element after the end of the set of `Thread` objects.

`ThreadPool::iterator find(Dyninst::LWP lwp)`
`ThreadPool::const_iterator find(Dyninst::LWP lwp) const`

The functions respectively return an `iterator` and a `const_iterator` that points to the `Thread` with a LWP of `lwp`. If `lwp` is not found in the thread list, then this function returns `end()`.

`size_t size() const`

This function returns the number of `Threads` in the `Process`.

`Process::ptr getProcess()`
`Process::const_ptr getProcess() const`

These functions respectively return a pointer or a const pointer to the `Process` that owns this `ThreadPool`.

`Thread::ptr getInitialThread()`
`Thread::const_ptr getInitialThread() const`

These functions respectively return a pointer or a const pointer to the initial `Thread` in a `Process`. The initial thread is the thread that started execution of the process (i.e., the thread that called `main`).

## 3.7. LibraryPool

A LibraryPool is a container representing the executable and set shared libraries (e.g., .dll and .so libraries) loaded into the target process' address space. A statically linked target process will only have a single executable, while a dynamically linked target process will have an executable and zero or more shared libraries.

The LibraryPool class contains iterators and search functions for operating on the set of libraries.

**LibraryPool Declared In:**
`PCProcess.h`

**LibraryPool Types:**

```
class iterator {
public:
     iterator();
     ~iterator();
     Library::ptr operator*() const;
     bool operator==(const iterator &i);
     bool operator!=(const iterator &i);
     LibraryPool::iterator operator++();
     LibraryPool::iterator operator++(int);
};
class const_iterator {
     const_iterator();
     ~const_iterator();
     Library::const_ptr operator*() const;
     bool operator==(const const_interator &i);
     bool operator!=(const const_iterator &i);
     LibraryPool::const_iterator operator++();
     LibraryPool::const_iterator operator++(int);
};
```

The `iterator` and `const_iterator` types of `LibraryPool` are respectively C++ iterators and const iterators over the set of libraries represented by the `LibraryPool`. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

**LibraryPool Member Functions:**

`LibraryPool::iterator begin()`
`LibraryPool::const_iterator begin() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the beginning of the set of `Library` objects.

`LibraryPool::iterator end()`
`LibraryPool::const_iterator end() const`

These functions respectively return an `iterator` and a `const_iterator` that point to the iterator element after the end of the set of `Library` objects.

`size_t size() const`

This function returns the number of elements in the library set.

`Library::ptr getExecutable()`
`Library::const_ptr getExecutable() const`

These functions respectively return a pointer or a const pointer to the `Library` object that represents the target process' executable.

`Library::ptr getLibraryByName(std::string name)`
`Library::const_ptr getLibraryByName(std::string name) const`

These functions respectively return a pointer or a const pointer to the `Library` object that with a file name equal to `name`. If no library is found then these functions respectively return `Library::ptr()` or `Library::const_ptr()`.

## 3.8. RegisterPool

The `RegisterPool` object represents a set of registers. It can be used to get or set all registers in a Thread at once. See the `Thread::getAllRegisters` and `Thread::setAllRegisters` functions. See Appendix A for more information about `MachRegister` and `MachRegisterVal`.

**RegisterPool Declared In:**
`PCProcess.h`

**RegisterPool Types:**

```
class iterator {
public:
      iterator();
      ~iterator();
      std::pair<MachRegister, MachRegisterVal> operator*() const;
      bool operator==(const iterator &i);
      bool operator!=(const iterator &i);
      RegisterPool::iterator operator++();
      RegisterPool::iterator operator++(int);
};
```

This is a C++ `iterator` over the set of registers contained in the registerPool. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

**RegisterPool Member Functions:**
`LibraryPool::iterator begin()`

This function returns an iterator that points to the beginning of the set of registers.

`LibraryPool::iterator end()`

This function returns an iterator that points element after the end of the set of registers.

`LibraryPool::iterator find(Dyninst::MachRegister r)`

This function returns an iterator that points to the element in the register pool that equals register `r`. If `r` is not found, then this function returns `end()`.

`size_t size() const`

This function returns the number of elements in the register set.

`Dyninst::MachRegisterVal& operator[](Dyninst::MachRegister r)`

This function returns a reference to the value associated with the register `r` in this register pool. It can be used to efficiently get and set the values of registers in this pool, or to fill the pool with new `MachRegister` objects.

## 3.9. AddressSet

The `AddressSet` class is a set container of `Process` and `Dyninst::Address` tuples, with each set element a `std::pair<Address, Process::ptr>`. `AddressSet` is used by the `ProcessSet` and `ThreadSet` classes for performing group operations on large numbers of processes. An `AddressSet` might, for example, represent the location of a symbol across numerous processes, or the location of a buffer in each process where data can be written or read.

The iteration interfaces of AddressSet resemble a C++ STL `std::multimap<Address, Process::ptr>`. When iterating all `Addresses` will appear in sequential order from smallest to largest.

**AddressSet Declared In:**

`ProcessSet.h`

**AddressSet Types:**

`AddressSet::ptr`
`AddressSet::const_ptr`

The `AddressSet::ptr` and `AddressSet::const_ptr` respectively represent a pointer and a const pointer to an AddressSet object. Both pointer types are reference counted and will cause the underlying `AddressSet` object to be cleaned when there are no more references.

```
typedef std::pair<Dyninst::Address, Process::ptr> value_type
class iterator {
public:
     iterator();
     ~iterator();
     value_type operator*() const;
     bool operator==(const iterator &i);
     bool operator!=(const iterator &i);
     AddressSet::iterator operator++();
     AddressSet::iterator operator++(int);
};
class const_iterator {
public:
     const_iterator();
     ~const_iterator();
     value_type operator*() const;
     bool operator==(const const_interator &i);
     bool operator!=(const const_iterator &i);
     AddressSet::const_iterator operator++();
     AddressSet::const_iterator operator++(int);
};
```

These are C++ iterators over the Address and Process pairs contained in the AddressSet. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

**AddressSet Static Member Functions:**

`static AddressSet::ptr newAddressSet()`

This function returns a new `AddressSet` that is empty.

```
static AddressSet::ptr newAddressSet(ProcessSet::const_ptr ps,
     Dyninst::Address addr)
```

This function returns a new `AddressSet` initialized with the elements from `ps` paired with the Address `addr`.

```
static AddressSet::ptr newAddressSet(ProcessSet::const_ptr ps,
     std::string library_name,
     Dyninst::Offset off)
```

This function returns a new `AddressSet` initialized with the elements from `ps`. The `Address` element for each process is calculated by looking up the load address of `library_name` in each `Process` and adding it to `off`.

**AddressSet Member Functions**

```
iterator begin()
const_iterator begin() const
```

These functions return an `iterator` that points to the first element in the `AddressSet`, or `end()` if the `AddressSet` is empty.

`iterator end()`
`const_iterator end() const`

These functions return an `iterator` that points to the element that comes after the final element in the `AddressSet`.

`iterator find(Dyninst::Address addr)`
`const_iterator find(Dyninst::Address addr) const`

These functions return an `iterator` that points to the first element in the `AddressSet` with an address of `addr`. They return `end()` if no element matches `addr`.

`iterator find(Dyninst::Address addr, Process::const_ptr proc)`
`const_iterator find(Dyninst::Address addr,`
`    Process::const_ptr proc) const`

These functions return an `iterator` that points to any element that has a process and address of `proc` and `addr`. It returns `end()` if no element matches.

`size_t count(Dyninst::Address addr) const`

This function returns the number of elements with address `addr`.

`size_t size() const`

This function returns the number of elements in the `AddressSet`.

`bool empty() const`

This function returns true if the `AddressSet` has zero elements and false otherwise.

`std::pair<iterator, bool> insert(Dyninst::Address addr,`
`    Process::const_ptr proc)`

This function inserts a new element into the `AddressSet` with `addr` and `proc` as its values. If another element with those values already exists, then no new element will be inserted. It returns an `iterator` that points to the new or existing element and a boolean value that is true if a new element was inserted and false otherwise.

`size_t insert(Dyninst::Address addr, ProcessSet::const_ptr ps)`

For every element in `ps`, this function inserts it and `addr` into the AddressSet. It returns the number of new elements created.

`void erase(iterator pos)`

This function removes the element pointed to by `pos` from the `AddressSet`.

`size_t erase(Process::const_ptr proc)`

This function removes every element with a process of `proc` from the `AddressSet`. It returns the number of elements removed.

`size_t erase(Dyninst::Address addr, Process::const_ptr proc)`

This function removes any element that has and address and process of `addr` and `proc` from the `AddressSet`. It returns the number of elements removed.

`void clear()`

This function erases all elements from the `AddressSet` leaving an `AddressSet` of size zero.

`iterator lower_bound(Dyninst::Address addr)`

This function returns an `iterator` pointing to the first element in the `AddressSet` that has an address greater than or equal to `addr`.

`iterator upper_bound(Dyninst::Address addr)`

This function returns an `iterator` pointing to the first element in the `AddressSet` that has an address greater than `addr`.

`std::pair<iterator, iterator> equal_range(Address addr) const`

This function returns a pair of `iterators`. The first `iterator` has the same value as the return of `lower_bound(addr)` and the second iterator has the same value as the return of `upper_bound(addr)`.

`AddressSet::ptr set_union(AddressSet::const_ptr aset)`

This function returns a new `AddressSet` whose elements are the set union of this `AddressSet` and `aset`.

`AddressSet::ptr set_intersection(AddressSet::const_ptr aset)`

This function returns a new `AddressSet` whose elements are the set intersection of this `AddressSet` and `aset`.

`AddressSet::ptr set_difference(AddressSet::const_ptr aset)`

This function returns a new `AddressSet` whose elements are the set difference of this `AddressSet` minus `aset`.

## 3.10. ProcessSet

The `ProcessSet` class is a set container for multiple `Process` objects. It shares many of the same operations as the `Process` class, but when an operation is performed on a ProcessSet it is done on every Process in the `ProcessSet`. On some systems, such as Blue Gene/Q, a `ProcessSet` can achieve better performance when repeating an operation across many target processes.

**ProcessSet Declared In:**

`ProcessSet.h`

**ProcessSet Types**

`ProcessSet::ptr`
`ProcessSet::const_ptr`

The `ptr` and `const_ptr` types are smart pointers to a `ProcessSet` object. When the last smart pointer to the `ProcessSet` is cleaned, then the underlying `ProcessSet` is cleaned.

`ProcessSet::weak_ptr`
`ProcessSet::const_weak_ptr`

The `weak_ptr` and `const_weak_ptr` are weak smart pointers to a `ProcessSet` object. Unlike regular smart pointers, weak pointers are not counted as references when determining whether to clean the `ProcessSet` object.

```
struct CreateInfo {
   std::string executable;
   std::vector<std::string> argv;
   std::vector<std::string> envp;
   std::map<int, int> fds;
   ProcControlAPI::err_t error_ret;
   Process::ptr proc;
}
struct AttachInfo {
   Dyninst::PID pid;
   std::string executable;
   ProcControlAPI::err_t error_ret;
   Process::ptr proc;
}
```

The `CreateInfo` and `AttachInfo` types are used by the `ProcessSet::createProcessSet` and `ProcessSet::attachProcessSet` functions when creating groups of processes.

```
class iterator {
public:
   iterator()
   ~iterator()
   Process::ptr operator*() const
   bool operator==(const iterator &i) const
   bool operator!=(const iterator &i) const
   ProcessSet::iterator operator++();
   ProcessSet::iterator operator++(int);
}
class const_iterator {
public:
   const_iterator()
   ~const_iterator()
   Process::const_ptr operator*() const
   bool operator==(const const_iterator &i) const
   bool operator!=(const const_iterator &i) const
   ProcessSet::const_iterator operator++();
   ProcessSet::const_iterator operator++(int);
}
```

These are C++ iterators over the `Process` pointers contained in the `ProcessSet`. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

```
struct write_t {
   void *buffer
   Dyninst::Address addr
   size_t size
   err_t err
   bool operator<(const write_t &w)
}
struct read_t {
   Dyninst::Address addr
   void *buffer
   size_t size
   err_t err
   bool operator<(const read_t &r)
}
```

The `write_t` and `read_t` types are used by `ProcessSet::readMemory` and `ProcessSet::writeMemory`.

**ProcessSet Static Member Functions**

`static ProcessSet::ptr newProcessSet()`

This function creates a new `ProcessSet` that is empty.

`static ProcessSet::ptr newProcessSet(Process::const_ptr proc)`

This function creates a new `ProcessSet` containing `proc`.

`static ProcessSet::ptr newProcessSet(ProcessSet::const_ptr pset)`

This function creates a new `ProcessSet` that is a copy of `pset`.

```
static ProcessSet::ptr newProcessSet(
      const std::set<Process::const_ptr> &procs)
```

This function creates a new `ProcessSet` containing every element from `procs`.

```
static ProcessSet newProcessSet(AddressSet::const_iterator ab,
      AddressSet::const_iterator ae)
```

This function creates a new `ProcessSet` containing the processes that are found within [`ab`, `ae`) of an `AddressSet`.

```
static ProcessSet::ptr createProcessSet(
      std::vector<CreateInfo> &cinfo)
```

This function creates a new `ProcessSet` by launching new processes. Each element in `cinfo` specifies an executable, arguments, environment and file descriptor mappings (with similar semantics to `Process::createProcess`), which are used to launch a new process.

Every successfully created `Process` will be added to a new `ProcessSet` that is returned by this function.

In addition, the `cinfo` vector will be updated so that each entry's `proc` field points to the `Process` created by that entry, and the `error_ret` entry will contain an error code for any process launch that failed.

`static ProcessSet::ptr attachProcessSet(`
`std::vector<AttachInfo> &ainfo)`

This function creates a new `ProcessSet` by attaching to existing processes. Each element in `ainfo` specifies a PID and executable (with similar semantics to `Process::attachProcess`), which are used to attach to the processes.

Every successfully attached `Process` will be added to a new `ProcessSet` that is returned by this function.

In addition, the `ainfo` vector will be updated so that each entry's `proc` field points to the `Process` attached by that entry, and the `error_ret` entry will contain an error code any process attach that failed.

**ProcessSet Member Functions**

`ProcessSet::ptr set_union(ProcessSet::ptr pset) const`

This function returns a new `ProcessSet` whose elements are a set union of this `ProcessSet` and `pset`.

`ProcessSet::ptr set_intersection(ProcessSet::ptr pset) const`

This function returns a new `ProcessSet` whose elements are a set intersection of this `ProcessSet` and `pset`.

`ProcessSet::ptr set_difference(ProcessSet::ptr pset) const`

This function returns a new `ProcessSet` whose elements are a set difference of this `ProcessSet` minus `pset`.

`iterator begin()`
`const_iterator begin() const`

These functions return iterators to the first element in the `ProcessSet`.

`iterator end()`
`const_iterator end() const`

These functions return iterators that come after the last element in the `ProcessSet`.

`iterator find(Process::const_ptr proc)`
`const_iterator find(Process::const_ptr proc) const`

These functions search a `ProcessSet` for the `Process` pointed to by `proc` and returns an iterator that points to that element. It returns `ProcessSet::end()` if no element is found.

`iterator find(Dyninst::PID pid)`
`const_iterator find(Dyninst::PID pid) const`

These functions search a `ProcessSet` for the Process `pointed` to by `proc` and returns an iterator that points to that element. It returns `ProcessSet::end()` if no element is found.

`bool empty() const`

This function returns true if the `ProcessSet` has zero elements, false otherwise.

`size_t size() const`

This function returns the number of elements in the `ProcessSet`.

`std::pair<iterator, bool> insert(Process::const_ptr proc)`

This function inserts `proc` into the `ProcessSet`. If `proc` already exists in the `ProcessSet`, then no change will occur. This function returns an iterator pointing to either the new or existing element and a boolean that is true if an insert happened and false otherwise.

`void erase(iterator pos)`

This function removes the element pointed to by `pos` from the `ProcessSet`.

`size_t erase(Process::const_ptr proc)`

This function searches the `ProcessSet` for `proc`, then erases that element from the `ProcessSet`. It returns 1 if it erased an element and 0 otherwise.

`void clear()`

This function erases all elements in the `ProcessSet`.

`ProcessSet::ptr getErrorSubset() const`

This function returns a new `ProcessSet` containing every `Process` from this `ProcessSet` that has a non-zero error code. Error codes are reset upon every `ProcessSet` API call, so this function shows which `Processes` had an error on the last `ProcessSet` operation.

```
void getErrorSubsets(std::map<ProcControlAPI::err_t,
    ProcessSet::ptr> &err_sets) const
```

This function returns a set of new `ProcessSets` containing every `Process` from this `ProcessSet` that has non-zero error codes, and grouped by error code. For each error code generated by the last `ProcessSet` API operation an element will be added to `err_sets`, and every `Process` that has the same error code will be added to the new `ProcessSet` associated with that error code.

```
bool anyTerminated() const;
bool allTerminated() const;
```

These functions respectively return true if any or all processes in this `ProcessSet` are terminated, and false otherwise.

```
bool anyExited() const;
bool allExited() const;
```

These functions respectively return true if any or all processes in this `ProcessSet` have exited normally, and false otherwise.

```
bool anyCrashed() const
bool allCrashed() const
```

These functions respectively return true if any or all processes in this `ProcessSet` have crashed normally, and false otherwise.

```
bool anyDetached();
bool allDetached();
```

These functions respectively return true if any or all processes in this `ProcessSet` have been detached, and false otherwise.

`bool anyThreadStopped();`
`bool allThreadStopped();`

These functions respectively return true if any or all threads in this `ProcessSet` are stopped, and false otherwise.

`bool anyThreadRunning();`
`bool allThreadRunning();`

These functions respectively return true if any or all threads in this `ProcessSet` are running, and false otherwise.

`ProcessSet::ptr getTerminatedSubset() const`

This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that is terminated.

`ProcessSet::ptr getExitedSubset() const`

This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that has exited normally.

`ProcessSet::ptr getCrashedSubset() const`

This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that has crashed.

`ProcessSet::ptr getDetachedSubset() const`

This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that is detached.

`ProcessSet::ptr getAllThreadRunningSubset() const`
`ProcessSet::ptr getAnyThreadRunningSubset() const`

This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that respectively has any or all threads running.

`ProcessSet::ptr getAllThreadStoppedSubset() const`
`ProcessSet::ptr getAnyThreadStoppedSubset() const`

This function returns a new `ProcessSet`, which is a subset of this `ProcessSet`, and contains every `Process` that respectively has any or all threads stopped.

`bool continueProcs()`

This function continues every thread in every process of this `ProcessSet`, similar to `Process::continueProc`. It returns true if every process was successfully continued and false otherwise.

`bool stopProcs()`

This function stops every thread in every process of this `ProcessSet`, similar to `Process::stopProc`. It returns true if every process was successfully stopped and false otherwise.

`bool detach(bool leaveStopped = true)`

This function detaches from every process in this `ProcessSet`, similar to `Process::detach`. It returns true if every process detach was successful and false otherwise.

If the `leaveStopped` parameter is set to `true`, and the processes in this `ProcessSet` are stopped, then the processes will be left in a stopped state after the detach.

`bool terminate()`

This function terminates every process in this `ProcessSet`, similar to `Process::terminate`. It returns true if every process was successfully terminated and false otherwise.

`bool temporaryDetach()`

This function does a temporary detach from every process in this `ProcessSet`, similar to `Process::temporaryDetach`. It returns true if every process was successfully detached and false otherwise.

`bool reAttach()`

This function reattaches to every process in this `ProcessSet`, similar to `Process::reAttach`. It returns true if every process was successfully reAttached and false otherwise.

`AddressSet::ptr mallocMemory(size_t sz) const`

This function allocates a block of memory of size `sz` in each process in this `ProcessSet`. The addresses of the allocations are returned in a new `AddressSet` object.

`bool mallocMemory(size_t size, AddressSet::ptr location)`

This function allocates a block of memory of size `sz` in each process in this `ProcessSet`. The memory will be allocated in each process based on the `Process/Address` pairs in `location`.

This function's behavior is undefined if `location` contains processes not included in this `ProcessSet`.

This function returns true if every allocation happened without error and false otherwise.

`bool freeMemory(AddressSet::ptr addrs) const`

This function frees memory allocated by Process::mallocMemory or `ProcessSet::mallocMemory`. The `AddressSet addrs` should contain a list of `Process/Address` pairs that point to the memory that should be freed.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every free happened without error and false otherwise.

```
bool readMemory(AddressSet::ptr addrs,
    std::multimap<Process::ptr, void *> &result,
    size_t size) const
```

This function reads memory from processes in this `ProcessSet`. `addrs` should contain the addresses to read memory from. `size` should be the amount of memory read from each process. The results of the memory reads will be returned by filling in the `result multimap`. Each process that is read from will have an entry in `result` along with a `malloc` allocated buffer containing the results of the read.

It is the ProcControlAPI user's responsibility to `free` the memory buffers returned by this function.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every read happened without error, and false otherwise.

```
bool readMemory(AddressSet::ptr addrs,
    std::map<void *, ProcessSet::ptr> &result,
    size_t size)
```

This function reads memory from processes in this `ProcessSet`. `addrs` should contain the addresses to read memory from. `size` should be the amount of memory to read from each process. The results of the memory reads will be aggregated together into the `result map`. If any two processes read equivalent byte-for-byte data, then those processes are grouped together in a new `ProcessSet` associated with a common `malloc` allocated buffer containing their memory contents.

It is the ProcControlAPI user's responsibility to `free` the memory buffers returned by this function.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every read happened without error, and false otherwise.

```
bool readMemory(std::multimap<Process::const_ptr, read_t> &addr)
```

This function reads memory from processes in this `ProcessSet`. The processes to read from are specified in the indexes of `addr`. The remote address, read size and local buffer are specified in the `read_t` elements of `addr`.

This function's behavior is undefined if `addr` contains processes not included in this `ProcessSet`.

This function returns true if every read happened without error, and false otherwise. If any read results in an error, then the `error_ret` field of the associated `addr` element will be set.

```
bool writMemory(AddressSet::ptr addrs,
    const void *buffer,
    size_t sz) const
```

This function will write the contents of `buffer` of size `sz` into the memory of each process at `addrs`.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every write happened without error, and false otherwise.

```
bool writeMemory(
    std::multimap<Process::const_ptr, write_t> &addrs) const
```

This function writes to the memory of each process in `addrs`. The processes to write to are specified as the indexes of `addrs`. The local memory buffer, buffer size, and target location are specified in the `write_t` element of `addrs`.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every write happened without error, and false otherwise. If any write results in an error, then the `error_ret` field of the associated `addr` element will be set.

```
bool addBreakpoint(AddressSet::ptr as, Breakpoint::ptr bp) const
```

This function inserts the `Breakpoint bp` into every process and address specified by as. It is similar to `Process::addBreakpoint`.

This function's behavior is undefined if `addrs` contains processes not included in this `ProcessSet`.

This function returns true if every breakpoint add happened without error, and false otherwise.

```
bool rmBreakpoint(AddressSet::ptr as, Breakpoint::ptr bp) const
```

The function removes the `Breakpoint bp` from each process at the locations specified in `as`. It is similar to `Process::rmBreakpoint`.

This function's behavior is undefined if `as` contains processes not included in this `ProcessSet`.

This function returns true if every breakpoint remove happened without error, and false otherwise.

```
bool postIRPC(const std::multimap<Process::const_ptr, IRPC::ptr>
    &rpcs) const
```

This function posts the `IRPC` objects specified in `rpcs` to their associated processes in the `multimap`. It is similar to `Process::postIRPC`.

This function's behavior is undefined if `rpcs` contains processes not included in this `ProcessSet`.

This function returns true if every post happened without error, and false otherwise.

```
bool postIRPC(IRPC::ptr irpc,
    std::multimap<Process::ptr, IRPC::ptr> *result = NULL)
```

This function makes a copy of `irpc` for each `Process` in this `ProcessSet` and posts it to that `Process`. If `result` is non-NULL, then the `multimap` will be filled with each newly created `IRPC` and the `Process` to which it was posted. It is similar to `Process::postIRPC`.

This function returns true if every post happened without error, and false otherwise.

```
bool postIRPC(IRPC::ptr irpc
     AddressSet::ptr addrs,
     std::multimap<Process::ptr, IRPC::ptr> *result = NULL)
```

This function makes a copy of `irpc` and posts it to each Process in `addrs` at the given `Address`. If `result` is non-NULL, then the `multimap` will be filled with each newly created `IRPC` and the `Process` to which it was posted. It is similar to `Process::postIRPC`.

This function's behavior is undefined if `rpcs` contains processes not included in this `ProcessSet`.

This function returns true if every post happened without error, and false otherwise.

## 3.11. ThreadSet

The `ThreadSet` class is a set container for `Thread` pointers. It has similar operations as `Thread`, and operations done on a `ThreadSet` affect every `Thread` in that `ThreadSet`. One some system, such as Blue Gene Q, using a `ThreadSet` is more efficient when doing the same operation across a large number of `Threads`.

**ThreadSet Declared In:**

ProcessSet.h

**ThreadSet Types:**

```
ThreadSet::ptr
ThreadSet::const_ptr
```

The `ptr` and `const_ptr` types are smart pointers to a `ThreadSet` object. When the last smart pointer to the `ThreadSet` is cleaned, then the underlying `ThreadSet` is cleaned. The `const_ptr` type is a const smart pointer.

```
ThreadSet::weak_ptr
ThreadSet::const_weak_ptr
```

The `weak_ptr` and `const_weak_ptr` are weak smart pointers to a `ThreadSet` object. Unlike regular smart pointers, weak pointers are not counted as references when determining whether to clean the `ThreadSet` object. The `const_weak_ptr` type is a const weak smart pointer.

```
class iterator {
public:
   iterator()
   ~iterator()
   Thread::ptr operator*() const
   bool operator==(const iterator &i) const
   bool operator!=(const iterator &i) const
   ThreadSet::iterator operator++();
   ThreadSet::iterator operator++(int);
}

class const_iterator {
public:
   const_iterator()
   ~const_iterator()
   Thread::const_ptr operator*() const
   bool operator==(const const_iterator &i) const
   bool operator!=(const const_iterator &i) const
   ThreadSet::const_iterator operator++();
   ThreadSet::const_iterator operator++(int);
}
```

These are C++ `iterators` over the Thread pointers contained in the ThreadSet. The behavior of `operator*`, `operator==`, `operator!=`, `operator++`, and `operator++(int)` match the standard behavior of C++ iterators.

**ThreadSet Static Member Functions**

`static ThreadSet::ptr newThreadSet()`

This function creates a new `ThreadSet` that is empty.

`static ThreadSet::ptr newThreadSet(Thread::ptr thr)`

This function creates a new `ThreadSet` that contains `thr`.

`static ThreadSet::ptr newThreadSet(const ThreadPool &threadp)`

This function creates a new `ThreadSet` that contains all of the `Threads` currently in `threadp`.

```
static ThreadSet::ptr newThreadSet (
    const std::set<Thread::const_ptr> &thrds)
```

This function creates a new `ThreadSet` that contains all of the threads in `thrds`.

`static ThreadSet::ptr newThreadSet(ProcessSet::ptr pset)`

This function creates a new `ThreadSet` that contains every live thread currently in every process in `pset`.

**ThreadSet Member Functions**

`ThreadSet::ptr set_union(ThreadSet::ptr tset) const`

This function returns a new `ThreadSet` whose elements are a set union of this `ThreadSet` and `tset`.

```
ThreadSet::ptr set_intersection(ThreadSet::ptr tset) const
```

This function returns a new `ThreadSet` whose elements are a set intersection of this `ThreadSet` and `tset`.

```
ThreadSet::ptr set_difference(ThreadSet::ptr tset) const
```

This function returns a new `ThreadSet` whose elements are a set difference of this `ThreadSet` minus `tset`.

```
iterator begin()
const_iterator begin() const
```

These functions return iterators to the first element in the `ThreadSet`.

```
iterator end()
const_iterator end() const
```

These functions return iterators that come after the last element in the `ThreadSet`.

```
iterator find(Thread::const_ptr thr)
const_iterator find(Thread::const_ptr thr) const
```

These functions search a `ThreadSet` for `thr` and returns an iterator pointing to that element. It returns `ThreadSet::end()` if no element is found

```
bool empty() const
```

This function returns true if the `ThreadSet` has zero elements and false otherwise.

```
size_t size() const
```

This function returns the number of elements in the `ThreadSet`.

```
std::pair<iterator, bool> insert(Thread::const_ptr thr)
```

This function inserts `thr` into the `ThreadSet`. If `thr` already exists in the `ThreadSet`, then no change will occur. This function returns an iterator pointing to either the new or existing element and a boolean that is true if an insert happened and false otherwise.

```
void erase(iterator pos)
```

This function removes the element pointed to by `pos` from the `ThreadSet`.

```
size_t erase(Thread::const_ptr thr)
```

This function searches the `ThreadSet` for `thr`, then erases that element from the `ThreadSet`. It returns 1 if it erased an element and 0 otherwise.

```
void clear()
```

This function erases all elements in the `ThreadSet`.

```
ThreadSet::ptr getErrorSubset() const
```

This function returns a new `ThreadSet` containing every `Thread` from this `ThreadSet` that has a non-zero error code. Error codes are reset upon every `ThreadSet` API call, so this function shows which `Threads` had an error on the last `ThreadSet` operation.

```
void getErrorSubsets(
    std::map<ProcControlAPI::err_t, ThreadSet::ptr> &err) const
```

This function returns a set of new `ThreadSets` containing every `Thread` from this `ThreadSet` that has non-zero error codes, and grouped by error code. For each error code generated by the last `ThreadSet` API operation an element will be added to `err`, and every `Thread` that has that error code will be added to the new `ThreadSet` associated with that error code.

`bool allStopped() const`
`bool anyStopped() const`

These functions respectively return true if any or all threads in this `ThreadSet` are stopped and false otherwise.

`bool allRunning() const`
`bool anyRunning() const`

These functions respectively return true if any or all threads in this `ThreadSet` are running and false otherwise.

`bool allTerminated() const`
`bool anyTerminated() const`

These functions respectively return true if any or all threads in this `ThreadSet` are terminated and false otherwise.

`bool allSingleStepMode() const`
`bool anySingleStepMode() const`

These functions respectively return true if any or all threads in this `ThreadSet` are running in single step mode, and false otherwise.

`bool allHaveUserThreadInfo() const`
`bool anyHaveUserThreadInfo() const`

These functions respectively return true if any or all threads in this `ThreadSet` have user thread information available and false otherwise.

`ThreadSet::ptr getStoppedSubset() const`

This function returns a new `ThreadSet`, which is a subset of this `ThreadSet`, and contains every `Thread` that is stopped.

`ThreadSet::ptr getRunningSubset() const`

This function returns a new `ThreadSet`, which is a subset of this `ThreadSet`, and contains every `Thread` that is running.

`ThreadSet::ptr getTerminatedSubset() const`

This function returns a new `ThreadSet`, which is a subset of this `ThreadSet`, and contains every `Thread` that is terminated.

`ThreadSet::ptr getSingleStepSubset() const`

This function returns a new `ThreadSet`, which is a subset of this `ThreadSet`, and contains every `Thread` that is in single step mode.

`ThreadSet::ptr getHaveUserThreadInfoSubset() const`

This function returns a new `ThreadSet`, which is a subset of this `ThreadSet`, and contains every `Thread` that has user thread information available.

`bool getStartFunctions(AddressSet::ptr result) const`

This function fills in the `AddressSet` pointed to by `result` with the address of every start function of each `Thread` in this `ThreadSet`. This information is only available on threads that have user thread information available.

This function return true if it succeeded for every Thread, and false otherwise.

`bool getStackBases(AddressSet::ptr result) const`

This function fills in the `AddressSet` pointed to by `result` with the address of every stack base of each `Thread` in this `ThreadSet`. This information is only available on threads that have user thread information available.

This function return true if it succeeded for every Thread, and false otherwise.

`bool getTLSs(AddressSet::ptr result) const`

This function fills in the `AddressSet` pointed to by `result` with the address of every thread-local storage region of each `Thread` in this `ThreadSet`. This information is only available on threads that have user thread information available.

This function return true if it succeeded for every Thread, and false otherwise.

`bool stopThreads() const`

This function stops every `Thread` in this `ThreadSet`. It is similar to `Thread::stopThread`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool continueThreads() const`

This function stops every `Thread` in this `ThreadSet`. It is similar to `Thread::continueThread`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool setSingleStepMode(bool v) const`

This function puts every `Thread` in this `ThreadSet` into single step mode if `v` is true. It clears single step mode if `v` is false. It is similar to `Thread::setSingleStepMode`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool getRegister(Dyninst::MachRegister reg,
    std::map<Thread::ptr, Dyninst::MachRegisterVal> &res) const
```

This function gets the value of register `reg` in every `Thread` in this `ThreadSet`. The collected values are returned in the `res` map, with each `Thread` mapped to the value of `reg` in that thread. It is similar to `Thread::getRegister`.

This function return true if it succeeded for every `Thread`, and false otherwise.

```
bool getRegister(Dyninst::MachRegister reg,
    std::map<Dyninst::MachRegisterVal, ThreadSet::ptr> &res)
    const
```

This function gets the value of register `reg` in every Thread in this `ThreadSet` and then aggregates all identical values together. The `res` map will contain an entry for each unique register value, and map that value to a new `ThreadSet` that contains every `Thread` that produced that register value. It is similar to `Thread::getRegister`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool setRegister(Dyninst::MachRegister reg,`
`const std::map<ThreadSet::const_ptr,`
`Dyninst::MachRegisterVal> &vals) const`

This function sets the value of register `reg` in each `Thread` in this `ThreadSet`. The value set in each thread is looked up in the `vals` map. It is similar to `Thread::setRegister`.

This function's behavior is undefined if it is passed a `Thread` that is not in this `ThreadSet`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool setRegister(Dyninst::MachRegister reg,`
`Dyninst::MachRegisterVal val) const`

This function sets the register `reg` to `val` in each `Thread` in this `ThreadSet`. It is similar to `Thread::setRegister`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool getAllRegisters(`
`std::map<Thread::ptr, RegisterPool> &results) const`

This function gets the values of every register in each `Thread` in this `ThreadSet`. The register values are returned as `RegisterPools` in the `results` map, with each `Thread` mapped to its `RegisterPool`. It is similar to `Thread::getAllRegisters`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool setAllRegisters(`
`const std::map<Thread::const_ptr, RegisterPool> &val) const`

This function sets the values of every register in each `Thread` in this `ThreadSet`. The register values are extracted from the `val` map, with each `Thread` specifying its register values via the map. This function is similar to `Thread::setAllRegisters`.

This function's behavior is undefined if it is passed a `Thread` that is not in this `ThreadSet`.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool postIRPC(const std::multimap<Thread::const_ptr,`
`IRPC::ptr> &rpcs) const`

This function posts an `IRPC` to every `Thread` in this `ThreadSet`. The `IRPC` to post to each `Thread` is specified in the `rpcs multimap`. This function is similar to Thread::postIRPC.

This function return true if it succeeded for every `Thread`, and false otherwise.

`bool postIRPC(IRPC::ptr irpc,`
`std::multimap<Thread::ptr, IRPC::ptr> *result = NULL)`

This function posts a copy of `irpc` to every `Thread` in this `ThreadSet`. If `result` is non-NULL, then the new `IRPC` objects are returned in the `result` multimap, with the `Thread` mapped to the `IRPC` that was posted there. This function is similar to `Thread::postIRPC`.

This function return true if it succeeded for every `Thread`, and false otherwise.

## 3.12. EventNotify

The EventNotify class is used to notify the user when ProcControlAPI is ready to deliver a callback function. EventNotify is a singleton class, which means only one instance of it is ever instantiated. See Section 2.2.3 for a high level description of notification.

**EventNotify Declared In:**
`PCProcess.h`

**EventNotify Types:**
`typedef void (*notify_cb_t)()`
This function signature is used for light-weight notification callback.

**EventNotify Related Global Functions:**
`EventNotify *evNotify()`
This function returns the singleton instance of the EventNotify class.

**EventNotify Member Functions:**
`int getFD()`
This function returns a file descriptor. ProcControlAPI will write a byte that will be available for reading on this file descriptor when a callback function is ready to be invoked. Upon seeing that a byte has been written to this file descriptor (likely via `select` or `poll`) the user should call the `Process::handleEvents` function. The user should never actually read the byte from this file descriptor; ProcControlAPI will handle clearing the byte after the callback function is invoked.

This function returns `-1` on error. Upon an error a subsequent call to `getLastError` returns details on the error.

`void registerCB(notify_cb_t cb)`
This function registers a light-weight callback function that will be invoked when a ProcControlAPI wishes to notify the user when a callback function is ready to be invoked. This light-weight callback may be called by a ProcControlAPI internal thread or from a signal handler; the user is encouraged to keep its implementation appropriately safe for these circumstances.

`void removeCB(notify_cb_t cb)`
This function removes a light-weight callback that was previously registered with `EventNotify::registerCB`. ProcControlAPI will no longer invoke the cb function after this function completes.

## 3.13. EventType

The `EventType` class represents a type of event. Each instance of an `Event` happening has one associated `EventType`, and callback functions can be registered against `EventTypes`. All `EventTypes` have an associate code—an integral value that identifies the `EventType`. Some `EventTypes` also have a time associated with them (`Pre`, `Post`, or

None)—describing when an Event may occur relative to the Code. For example, an EventType with a code of Exit and a time of Pre (written as pre-exit) would be associated with an Event that occurs just before a process exits and its address space is cleaned. An EventType with code Exit and a time of Post would be associated with an Event that occurs after the process exits and the address space is cleaned.

When using EventTypes to register for callback functions a special time value of Any can be used. This signifies that the callback function should trigger for both Pre and Post time events. ProcControlAPI will never deliver an Event that has an EventType with time code Any.

More details on Events and EventTypes can be found in Section 2.2.1.

**EventType Types:**

```
typedef enum {
     Pre = 0,
     Post,
     None,
     Any
} Time;

typedef int Code;
```

The Time and Code types are respectively used to describe the time and code values of an EventType.

**EventType Constants:**

```
static const int Error = -1
static const int Unset = 0
static const int Exit = 1
static const int Crash = 2
static const int Fork = 3
static const int Exec = 4
static const int ThreadCreate = 5
static const int ThreadDestroy = 6
static const int Stop = 7
static const int Signal = 8
static const int LibraryLoad = 9
static const int LibraryUnload = 10
static const int Bootstrap = 11
static const int Breakpoint = 12
static const int RPC = 13
static const int SingleStep = 14
static const int Library = 15
static const int MaxProcCtrlEvent = 1000
```

These constants describe possible values for an EventType's code. The Error and Unset codes are for handling error cases and should not be used for callback functions or be associated with Events.

The EventType codes were implemented as an integer (rather than an enum) to allow users to create custom EventTypes. Any custom EventType should begin at the MaxProcCtrlEvent value, all smaller values are reserved by ProcControlAPI.

**EventType Related Types:**

```
struct eventtype_cmp {
    bool operator()(const EventType &a, const EventType &b);
}
```

This type defines a less-than comparison function for EventTypes. While a comparison of EventTypes does not have a semantic meaning, this can be useful for inserting EventTypes into maps or other STL data structures.

**EventType Member Functions:**

`EventType(Code e)`

Constructs an EventType with the given code and a time of Any.

`EventType(Time t, Code e)`

Constructs an EventType with the given time and code values.

`EventType()`

Constructs an EventType with an Unset code and None time value.

`Code code() const`

Returns the code value of the EventType.

`Time time() const`

Returns the time value of the EventType.

`std::string name() const`

Returns a human readable name for this EventType.

## 3.14. Event

The Event class represents an instance of an event happening. Each Event has an EventType that describes the event and pointers to the Process and Thread that the event occurred on.

The Event class is an abstract class that is never instantiated. Instead, ProcControlAPI will instantiate children of the Event class, each of which add information specific to the EventType. For example, an Event representing a thread creation will have an EventType of ThreadCreate and can be cast into an EventNewThread for specific information about the new thread. The specific events that are instantiated from Event are described in the Section 3.15.

An event that occurs on a running thread may cause the process, thread, or neither to stop running until the event has been handled. The specifics of what is stopped can change between different event types and operating systems. Each Event describes whether it stopped the associated process or thread with a SyncType field. The values of this field can be async (the event stopped neither the process nor thread), sync_thread (the event stopped its thread), or `sync_process` (the event stopped all threads in the process). A callback function can choose how to resume or stop a process or thread using its return value (see Section 2.2.2).

More details on `Event` can be found in Section 2.2.1.

**Event Declared In:**

`Event.h`

**Event Types:**

```
typedef enum {
     unset,
     async,
     sync_thread,
     sync_process
} SyncType
```

The `SyncType` type is used to describe how a process or thread is stopped by an `Event`. See the above explanation for more details.

**Event Member Functions:**

`Thread::const_ptr getThread() const`

This function returns a const pointer to the `Thread` object that represents the thread this event occurred on.

`Process::const_ptr getProcess() const`

This function returns a const pointer to the `Process` object that represents the process this event occurred on.

`EventType getEventType() const`

This function returns the `EventType` associated with this `Event`.

`SyncType getSyncType() const`

This function returns the `SyncType` associated with this `Event`.

`std::string name() const`

This function returns a human readable name for this Event.

```
EventTerminate::ptr getEventTerminate()
EventTerminate::const_ptr getEventTerminate() const

EventExit::ptr getEventExit()
EventExit::const_ptr getEventExit() const

EventCrash::ptr getEventCrash()
EventCrash::const_ptr getEventCrash() const

EventForceTerminate::ptr getEventForceTerminate()
EventForceTerminate::const_ptr getEventForceTerminate() const

EventExec::ptr getEventExec()
EventExec::const_ptr getEventExec() const

EventStop::ptr getEventStop()
EventStop::const_ptr getEventStop() const

EventBreakpoint::ptr getEventBreakpoint()
EventBreakpoint::const_ptr getEventBreakpoint() const

EventNewThread::ptr getEventNewThread()
EventNewThread::const_ptr getEventNewThread() const

EventNewUserThread::ptr getEventNewUserThread()
EventNewUserThread::const_tr getEventNewUserThread() const

EventNewLWP::ptr getEventNewLWP()
EventNewLWP::const_ptr getEventNewLWP() const

EventThreadDestroy::ptr getEventThreadDestroy()
EventThreadDestroy::const_ptr getEventThreadDestroy() const

EventUserThreadDestroy::ptr getEventUserThreadDestroy()
EventUserThreadDestroy::const_ptr getEventUserThreadDestroy()
const

EventLWPDestroy::ptr getEventLWPDestroy()
EventLWPDestroy::const_ptr getEventLWPDestroy() const

EventFork::ptr getEventFork()
EventFork::const_ptr getEventFor() const

EventSignal::ptr getEventSignal()
EventSignal::const_ptr getEventSignal() const
```

```
EventRPC::ptr getEventRPC()
EventRPC::const_ptr getEventRPC() const
```

```
EventSingleStep::ptr getEventSingleStep()
EventSingleStep::const_ptr getEventSingleStep() const
```

```
EventLibrary::ptr getEventLibrary()
EventLibrary::const_ptr getEventLibrary() const
```

These functions serve as a form of `dynamic_cast`. They cast the `Event` into a child type and return the result of that cast. If the `Event` object is not of the appropriate type for the given function, then they return a shared pointer `NULL` equivalent (`ptr()` or `const_ptr()`).

For example, if an `Event` was an instance of an `EventRPC`, then the `getEventRPC()` function would cast it to `EventRPC` and return the resulting value.

## 3.15.Event Child Classes

The `Event` class is an abstract parent class, while the classes listed in this section are the child classes that are actually instantiated. Given an `Event` object passed to a callback function, a ProcControlAPI user can inspect the `Event`'s `EventType` and cast it to the appropriate child class listed below.

Note that each child class inherits the member functions described in the Event class in Section 3.14.

**Common Types:**

```
<EventChildClassHere>::ptr
<EventChildClassHere>::const_ptr
```

These types are common to all `Event` children classes. Rather than repeat them for each class, they are listed once here for brevity.

The `ptr` and `const_ptr` respectively represent a pointer and a const pointer to an `Event` child class. Both pointer types are reference counted and will cause the underlying object will be cleaned when there are no more references.

### 3.15.1. EventTerminate

The `EventTerminate` class is a parent class for `EventExit` and `EventCrash`. It is never instantiated by ProcControlAPI and simply serves as a place-holder type for a user to deal with process termination without dealing with the specifics of whether a process exited properly or crashed.

**Associated EventType Codes:**
`Exit, Crash and ForceTerminate`

### 3.15.2. EventExit

An `EventExit` triggers when a process performs a normal exit (e.g., calling the `exit` function or returning from `main`). The process that exited is referenced with `Event`'s `getProcess` function.

An `EventExit` may be associated with an `EventType` of pre-exit or post-exit. Pre-exit means the process has not yet cleaned up its address space, and thus memory can still be read or written. Post-exit means the process has cleaned up its address space, memory is no longer accessible.

**Associated EventType Code:**
`Exit`

**EventExit Member Functions:**
`int getExitCode() const`
This function returns the process' exit code.

### 3.15.3. EventCrash

An `EventCrash` triggers when a process performs an abnormal exit (e.g., crashing on a memory violation). The process that crashed is referenced with `Event`'s `getProcess` function.

An `EventCrash` may be associated with an `EventType` of pre-crash or post-crash. Pre-crash means the process has not yet cleaned up its address space, and thus memory can still be read or written. Post-crash means the process has cleaned up its address space, memory is no longer accessible.

**Associated EventType Code:**
`Crash`

**EventCrash Member Functions:**
`int getTermSignal() const`
This function returns the signal that caused the process to crash.

### 3.15.4. EventForceTerminate

An `EventForceTerminate` triggers when a process is forcefully terminated via the `Process::terminate` function. When the callback is delivered for this event, the address space of the corresponding process will no longer be available.

**Associated EventType Code:**
`ForceTerminate`

**EventForceTerminate Member Functions:**
`int getTermSignal() const`
This function returns the signal that was used to terminate the process.

### 3.15.5. EventExec

An `EventExec` triggers when a process performs a UNIX-style `exec` operation. An `EventType` of post-Exec means the process has completed the `exec` and setup its new address space. An EventType of pre-Exec means the process has not yet torn down its old address space.

**Associated EventType Code:**
`Exec`

**EventExec Member Functions:**
`std::string getExecPath() const`
This function returns the file path to the process' new executable.

### 3.15.6. EventStop

An `EventStop` is triggered when a process is stopped by a non-ProcControlAPI source. On UNIX based systems, this is triggered by receipt of a SIGSTOP signal.

Unlike most other events, an `EventStop` will explicitly move the associated thread or process (see the `Event`'s `SyncType` to tell which) to a stopped state. Returning `cbDefault` from a callback function that has received `EventStop` will leave the target process in a stopped state rather than restore it to the pre-event state.

**Associated EventType Code:**
`Stop`

### 3.15.7. EventBreakpoint

An `EventBreakpoint` triggers when the target process encounters a breakpoint inserted by the ProcControlAPI (see Section 3.4).

Similar to EventStop, `EventBreakpoint` will explicitly move the thread or process to a stopped state. Returning `cbDefault` from a callback function that has received `EventBreakpoint` will leave the target process in a stopped state rather than restore it to the pre-event state.

**Associated EventType Code:**
`Breakpoint`

**EventBreakpoint Member Functions:**
`Dyninst::Address getAddress() const`
This function returns the address at which the breakpoint was hit.

`void getBreakpoints(std::vector<Breakpoint::const_ptr> &b) const`
This function returns a vector of pointers to the `Breakpoints` that were hit. Since it is possible to insert multiple `Breakpoints` at the same location, it is possible for this function to return more than one `Breakpoint`.

### 3.15.8. EventNewThread

An `EventNewThread` triggers when a process spawns a new thread. The `Event` class' `getThread` function returns the original `Thread` that performed the spawn operation, while `EventNewThread`'s `getNewThread` returns the newly created `Thread`.

This event is never instantiated by ProcControlAPI and simply serves as a place-holder type for a user to deal with thread creation without having to deal with the specifics of LWP and user thread creation.

A callback function that receives an `EventNewThread` can use the two field form of `Process::cb_ret_t` (see Sections 2.2.2 and 3.1) to control the parent and child thread.

**Associated EventType Codes:**

`ThreadCreate, UserThreadCreate, LWPCreate`

**EventNewThread Member Functions:**

`Thread::const_ptr getNewThread() const`

This function returns a const pointer to the `Thread` object that represents the newly spawned thread.

### 3.15.9. EventNewUserThread

An `EventNewUserThread` triggers when a process spawns a new user-level thread. The `Event` class' `getThread` function returns the original `Thread` that performed the spawn operation. This thread may have already been created if the platform supports the `EventNewLWP` event. If not, `the getNewThread function` returns the newly created `Thread`.

A callback function that receives an `EventNewThread` can use the two field form of `Process::cb_ret_t` (see Sections 2.2.2 and 3.1) to control the parent and child thread.

**Associated EventType Code:**

`UserThreadCreate`

**EventNewThread Member Functions:**

`Thread::const_ptr getNewThread() const`

This function returns a const pointer to the `Thread` object that represents the newly spawned thread or the corresponding thread, if the thread has already been created.

### 3.15.10. EventNewLWP

An `EventNewLWP` triggers when a process spawns a new LWP. The `Event` class' `getThread` function returns the original `Thread` that performed the spawn operation, while `EventNewThread`'s `getNewThread` returns the newly created `Thread`.

A callback function that receives an `EventNewThread` can use the two field form of `Process::cb_ret_t` (see Sections 2.2.2 and 3.1) to control the parent and child thread.

**Associated EventType Code:**
`LWPCreate`

**EventNewThread Member Functions:**
`Thread::const_ptr getNewThread() const`

This function returns a const pointer to the `Thread` object that represents the newly spawned thread.

### 3.15.11. EventThreadDestroy

An `EventThreadDestroy` triggers when a thread exits. Event's `getThread` member function returns the thread that exited.

This event is never instantiated by ProcControlAPI and simply serves as a place-holder type for a user to deal with thread destruction without having to deal with the specifics of LWP and user thread destruction.

**Associated EventType Codes:**
`ThreadDestroy, UserThreadDestroy, LWPDestroy`

### 3.15.12. EventUserThreadDestroy

An `EventUserThreadDestroy` triggers when a thread exits. Event's `getThread` member function returns the thread that exited.

If the platform also supports `EventLWPDestroy` events, this event will precede an `EventLWPDestroy` event.

**Associated EventType Code:**
`UserThreadDestroy`

### 3.15.13. EventLWPDestroy

An `LWPThreadDestroy` triggers when a thread exits. Event's `getThread` member function returns the thread that exited.

**Associated EventType Code:**
`LWPDestroy`

### 3.15.14. EventFork

An `EventFork` triggers when a process performs a UNIX-style fork operation. The process that performed the initial fork is returned via Event's `getProcess` member function, while the newly created process can be found via `EventFork`'s `getChildProcess` member function.

**Associated EventType Code:**
`Fork`

**EventFork Member Functions:**
`Process::const_ptr getChildProcess() const`

This function returns a pointer to the `Process` object that represents the newly created child process.

### 3.15.15. EventSignal

An `EventSignal` triggers when a process receives a UNIX style signal.

**Associated EventType Code:**
`Signal`

**EventSignal Member Functions:**
`int getSignal() const`

This function returns the signal number that triggered the `EventSignal`.

### 3.15.16. EventRPC

An EventRPC triggers when a process or thread completes a ProcControlAPI iRPC (see Sections 2.3 and 3.5). When a callback function receives an EventRPC, the memory and registers that were used by the iRPC can still be found in the address space and thread that the iRPC ran on. Once the callback function completes, the registers and memory are restored to their original state.

**Associated EventType Code:**
`RPC`

**EventRPC Member Functions:**
`IRPC::const_ptr getIRPC() const`

This function returns a const pointer to the IRPC object that completed.

### 3.15.17. EventSingleStep

An `EventSingleStep` triggers when a thread, which was put in single-step mode by `Thread::setSingleStep`, completes a single step operation. The `Thread` will remain in single-step mode after completion of this event (presuming it has not be explicitly disabled by `Thread::setSingleStep`).

**Associated EventType Code:**
`SingleStep`

### 3.15.18. EventLibrary

An `EventLibrary` triggers when the process either loads or unloads a shared library. ProcControlAPI will not trigger an `EventLibrary` for library unloads associated with a

Process being terminated, and it will not trigger `EventLibrary` for library loads that happened before a ProcControlAPI attach operation.

It is possible for multiple libraries to be loaded or unloaded at the same time. In this case, an `EventLibrary` will contain multiple libraries in its load and unload sets.

**Associated EventType Code:**
`Library`

**EventLibrary Member Functions:**

`const std::set<Library::ptr> &libsAdded() const`

This function returns the set of Library objects that were loaded into the target process' address space. The set will be empty if no libraries were loaded.

`const std::set<Library::ptr> &libsRemoved() const`

This function returns the set of libraries that were unloaded from the target process' address space. The set will be empty if no libraries were unloaded.

### 3.15.19. EventPreSyscall, EventPostSyscall

An `EventPreSyscall` is triggered when a thread enters a system call, provided that the thread is in system call tracing mode. An `EventPostSyscall` is triggered when a system call returns. These are both children of `EventSyscall`, which provides all the relevant methods.

**Associated EventType Code:**
`Syscall`

**EventPreSyscall and EventPostSyscall Member Functions:**

`Dyninst::Address getAddress() const`

This function returns the address where the system call occurred.

`MachSyscall getSyscall() const`

This function returns information about the system call. See Appendix B for information about the `MachSyscall` class.

# Appendix A. Registers

This appendix describes the `MachRegister` interface, which is used for accessing registers in ProcControlAPI. The entire definition of `MachRegister` contains more register names than are listed here; this appendix only lists the registers that can be accessed through ProcControlAPI.

An instance of `MachRegister` is defined for each register ProcControlAPI can name. These instances live inside a namespace that represents the register's architecture. For example, we can name a register from an AMD64 machine with `Dyninst::x86_64::rax` or a register from a Power machine with `Dyninst::ppc32::r1`.

All functions, types, and objects listed below are part of the C++ namespace `Dyninst`.

**Declared In:**

`dyn_regs.h`

**Related Types:**

```
typedef unsigned long MachRegisterVal
```

The `MachRegisterVal` type is used to represent the contents of a register. If a register's contents are smaller than `MachRegisterVal`, then it will be up cast into a `MachRegisterVal`.

```
typedef enum {
     Arch_none,
     Arch_x86,
     Arch_x86_64,
     Arch_ppc32,
     Arch_ppc64
} Architecture
```

The `Architecture` enum represents a system's architecture.

**Related Global Functions**

```
unsigned int getArchAddressWidth(Architecture arch)
```

Returns the size of a pointer, in bytes, on the given architecture, `arch`.

**MachRegister Static Member Functions**

```
MachRegister getPC(Dyninst::Architecture arch)
MachRegister getFramePointer(Dyninst::Architecture arch)
MachRegister getStackPointer(Dyninst::Architecture arch)
```

These functions respectively return the register that represents the program counter, frame pointer, or stack pointer for the given architecture. If an architecture does not support a frame pointer (ppc32 and ppc64) then getFramePointer returns an invalid register.

### MachRegister Member Functions

`MachRegister getBaseRegister() const`

This function returns the largest register that may alias with the given register. For example, `getBaseRegister` on `x86_64::ah` returns `x86_64::rax`.

`Architecture getArchitecture() const`

This function returns the `Architecture` for this register.

`bool isValid() const`

This function returns true if this register is valid. Some API functions may return invalid registers upon error.

```
MachRegisterVal getSubRegVal(
    MachRegister subreg,
    MachRegisterVal orig)
```

Given a value for this register, `orig`, and a smaller aliased register, `subreg`, then this function returns the value of the aliased register. For example, if this function were called on `x86::eax` with `subreg` as `x86::al` and an `orig` value of `0x11223344`, then it would return `0x44`.

`const char *name() const`

This function returns a human readable name that identifies this register.

`unsigned int size() const`

This function returns the size of the register in bytes.

`signed int val() const`

This function returns a unique integer that represents this register. This can be useful for writing switch statements that take a MachRegister. The unique integers for a MachRegister can be found by preceding the register object name with an ‘i'. For example, a switch statement for `MachRegister`, `reg`, could be written as:

```
switch (reg.val()) {
     case x86_64::irax:
     case x86_64::irbx:
     case x86_64::ircx:
     …
}
```

```
bool isPC() const
bool isFramePointer() const
bool isStackPointer() const
```

These functions respectively return true if the register is the program counter, frame pointer, or stack pointer for its architecture. They return false otherwise.

### MachRegister Objects

The following list describes instances of MachRegister that can be passed to ProcControlAPI. These can be named by prepending the namespace to the listed names, e.g., `x86::eax`.

`namespace x86`

| | | |
|---|---|---|
| eax | edi | fs |
| ebx | oeax | gs |
| ecx | eip | ss |
| edx | flags | fsbase |
| ebp | cs | gsbase |
| esp | ds | |
| esi | es | |

`namespace x86_64`

| | | |
|---|---|---|
| rax | r9 | flags |
| rbx | r10 | cs |
| rcx | r11 | ds |
| rdx | r12 | es |
| rbp | r13 | fs |
| rsp | r14 | gs |
| rsi | r15 | ss |
| rdi | orax | fsbase |
| r8 | rip | gsbase |

`namespace ppc32`

| | | |
|---|---|---|
| r0 | r13 | r26 |
| r1 | r14 | r27 |
| r2 | r15 | r28 |
| r3 | r16 | r29 |
| r4 | r17 | r30 |
| r5 | r18 | r31 |
| r6 | r19 | fpscw |
| r7 | r20 | lr |
| r8 | r21 | cr |
| r9 | r22 | xer |
| r10 | r23 | ctr |
| r11 | r24 | pc |
| r12 | r25 | msr |

`namespace ppc64`

| | | |
|---|---|---|
| r0 | r9 | r18 |
| r1 | r10 | r19 |
| r2 | r11 | r20 |
| r3 | r12 | r21 |
| r4 | r13 | r22 |
| r5 | r14 | r23 |
| r6 | r15 | r24 |
| r7 | r16 | r25 |
| r8 | r17 | r26 |

| | | |
|---|---|---|
| r27 | r31 | xer |
| r28 | fpscw | ctr |
| r29 | lr | pcmsr |
| r30 | cr | |

`namespace aarch64`

| | | |
|---|---|---|
| x0 | x23 | q15 |
| x1 | x24 | q16 |
| x2 | x25 | q17 |
| x3 | x26 | q18 |
| x4 | x27 | q19 |
| x5 | x28 | q20 |
| x6 | x29 | q21 |
| x7 | x30 | q22 |
| x8 | q0 | q23 |
| x9 | q1 | q24 |
| x10 | q2 | q25 |
| x11 | q3 | q26 |
| x12 | q4 | q27 |
| x13 | q5 | q28 |
| x14 | q6 | q29 |
| x15 | q7 | q30 |
| x16 | q8 | q31 |
| x17 | q9 | sp |
| x18 | q10 | pc |
| x19 | q11 | pstate |
| x20 | q12 | fpcr |
| x21 | q13 | fpsr |
| x22 | q14 | |

# Appendix B. System Calls

The MachSyscall class, found in MachSyscall.h, represents system calls in a platform-independent manner. Currently, syscall events are only supported on Linux.

**MachSyscall Methods**

`SyscallIDPlatform num() const`

Returns the platform-specific syscall number

`SyscallName name() const`

Returns the name of the system call (e.g. "getpid")

`bool operator==(const MachSyscall&) const`

Equality operator. Two system calls are equal if they are for the same platform and have the same syscall number.

```
static MachSyscall makeFromPlatform(Platform,
SyscallIDIndependent)
```

Constructs a MachSyscall from a Platform and a platform-independent ID (e.g. dyn_getpid). The platform-independent syscall IDs may be found in dyn_syscalls.h.

## Appendix C. Known Issues

Prior to Linux 2.6.38, some kernels allowed the debug interface to return multiple pending signals without receiving an explicit debugger continue. Proccontrol's architecture relies on receiving a single debugger event for each continue that it issues except at exit time. This can cause an unrecoverable assertion. We have only observed this behavior when a process is receiving signals both from itself and from another process, and we have only observed it when the self-signaling behavior is a breakpoint. This behavior does not occur with 2.6.38 and subsequent kernels, and it has not been observed on any kernels with utrace support (which covers all RedHat kernels that would otherwise be affected by this kernel bug).

Library load/unload, user-level thread creation/destruction, inferior RPCs, and user-inserted breakpoints can all cause self-signaling, and a process's children exiting or stopping will cause the parent to be signaled as well. While we cannot provide a general prescription for avoiding this bug (other than upgrading to an unaffected kernel), the above should suggest strategies for reducing the likelihood you will be affected by it.