Paradyn Parallel Performance Tools

ProcControlAPI

Developer’s Guide

Beta 1

March 2011

Computer Science Department

University of Wisconsin-Madison

Madison, WI 53711

Computer Science Department

University of Maryland

College Park, MD 20742

Email: bugs@dyninst.org

Web: www.dyninst.org

[1. Introduction 1](#_Toc274137360)

[1.1. Simple Example 1](#_Toc274137361)

[2. Important Concepts 4](#_Toc274137362)

[2.1. Processes and Threads 4](#_Toc274137363)

[2.2. Callbacks 4](#_Toc274137364)

[2.2.1. Events 4](#_Toc274137365)

[2.2.2. Callback Functions 6](#_Toc274137366)

[2.2.3. Callback Delivery 7](#_Toc274137367)

[2.3. iRPCs 7](#_Toc274137368)

[2.4. Memory Management 8](#_Toc274137369)

[3. API Reference 9](#_Toc274137370)

[3.1. Process 9](#_Toc274137371)

[3.2. Thread 16](#_Toc274137372)

[3.3. Library 20](#_Toc274137373)

[3.4. Breakpoint 21](#_Toc274137374)

[3.5. IRPC 22](#_Toc274137375)

[3.6. ThreadPool 23](#_Toc274137376)

[3.7. LibraryPool 25](#_Toc274137377)

[3.8. RegisterPool 27](#_Toc274137378)

[3.9. EventNotify 27](#_Toc274137379)

[3.10. EventType 28](#_Toc274137380)

[3.11. Event 30](#_Toc274137381)

[3.12. Event Child Classes 33](#_Toc274137382)

[3.12.1. EventTerminate 33](#_Toc274137383)

[3.12.2. EventExit 33](#_Toc274137384)

[3.12.3. EventCrash 34](#_Toc274137385)

[3.12.4. EventExec 34](#_Toc274137386)

[3.12.5. EventStop 35](#_Toc274137387)

[3.12.6. EventBreakpoint 35](#_Toc274137388)

[3.12.7. EventNewThread 35](#_Toc274137389)

[3.12.8. EventThreadDestroy 37](#_Toc274137390)

[3.12.9. EventFork 37](#_Toc274137391)

[3.12.10. EventSignal 38](#_Toc274137392)

[3.12.11. EventRPC 38](#_Toc274137393)

[3.12.12. EventSingleStep 38](#_Toc274137394)

[3.12.13. EventLibrary 38](#_Toc274137395)

[Appendix A. Registers 40](#_Toc274137396)

# 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 will assume 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.

## 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.
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.

#include "Process.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

1. 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.
2. 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.
3. The Event class has several child classes, one of which is EventNewThread. We start by casting the Event into a EventNewThread and then extract information about the new thread from the EventNewThread.
4. 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).
5. 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.
6. 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.

# 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.

## 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.

## 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.

### 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** |
| 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 3.11.

### 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.

### 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.

## 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 and 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 synchronous or asynchronous. If a synchronous iRPC is posted to any Process, then calls to Process::handleEvents will block until the synchronous iRPC is completed.

## 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.

# 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; )

## 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 Defined In:

Process.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 cb\_action\_t {

cbDefault,

cbThreadContinue,

cbThreadStop,

cbProcContinue,

cbProcStop

}

struct 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 will return 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 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.

Process Static Member Functions:

static Process::ptr createProcess(

std::string executable,

const std::vector<std::string> &argv,

const std::vector<std::string> &envp)

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 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 will return 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 will return 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 will return 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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return 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 will return 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 will return 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 will return 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.

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

bool hasRunningThread() const

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

bool allThreadsStopped() const

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

bool allThreadsRunning() const

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

bool allThreadsRunningWhenAttached() const

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

bool continueProc()

This function will move all threads in the target process into the running state. This function will return 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 will return details on the error.

bool stopProc()

This function will move all threads in the target process into the stopped state. This function will return 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 will return details on the error.

bool detach()

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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return details on the 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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return 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.

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 will return the start address of memory that was allocated and 0 otherwise. Upon an error a subsequent call to getLastError will return 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 will return true, otherwise it will return false. Upon an error a subsequent call to getLastError will return 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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return 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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return details on the error.

bool addBreakpoint(

Dyninst::Address addr,

Breakpoint::ptr bp) const

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

bool rmBreakpoint(

Dyninst::Address addr,

Breakpoint::ptr bp) const

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

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

bool postIRPC(IRPC::ptr irpc) const

This function posts the given irpc to the Process. ProcControlAPI will select 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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return details on the error.

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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return details on the error.

## Thread

The Thread class represents a single thread of execution in the target process. Any Process will have at least one Thread, and multi-threaded target processes may have more. Each Thread will have 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 Defined In:

Process.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 will cause the underlying Thread object to be cleaned when there are no more references. ProcControlAPI will maintain 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 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 will return true, otherwise it will return false. Upon an error a subsequent call to getLastError will return 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 will return true, otherwise it will return false. Upon an error a subsequent call to getLastError will return 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 will return true, otherwise it will return false. Upon an error a subsequent call to getLastError will return 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 will return true, otherwise it will return false. Upon an error a subsequent call to getLastError will return 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 will return true, otherwise it will return false. Upon an error a subsequent call to getLastError will return 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 will return true, otherwise it will return false. Upon an error a subsequent call to getLastError will return 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.l

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 0 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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return 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 will return true on success and false on error. Upon an error a subsequent call to getLastError will return 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.

## 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.

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 will return the load address of the code region and Library::getDataLoadAddress will return the load address of the data region. On non-AIX systems this function returns 0.

## 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 3.12.7).

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.

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 Static Functions

Breakpoint::ptr newBreakpoint()

This function creates a new 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 breakpoint. Upon resumption after executing this Breakpoint, control will resume at the address specified by the ctrl\_to parameter.

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.

## 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 Defined In:

Process.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 async = false)

IRPC::ptr createIRPC(

void \*binary\_blob,

unsigned int size,

Dyninst::Address addr,

bool async = 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 async parameter is true then the IRPC will be asynchronous, otherwise it will be synchronous.

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 will return 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 will return it. Otherwise this function returns 0.

## 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 will return all Threads.

ThreadPool Defined In:

Process.h

ThreadPool Types:

class iterator {

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 {

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).

## 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 Defined In:

Process.h

LibraryPool Types:

class iterator {

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().

## 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 Defined In:

Process.h

RegisterPool Types:

class iterator{

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);

};

The iterator is a C++ over the set of registers represented 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.

## 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 Defined In:

Process.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 will return -1 on error. Upon an error a subsequent call to getLastError will return 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.

## 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.

## 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 3.12.

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 Defined 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.

## 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 3.11.

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.

### 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

### 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.

### 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.

### 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.

### 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.

### 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

### 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.

### EventNewThread

An EventNewThread triggers when a process spawns a new thread. The Event class’ getThread function will return the original Thread that performed the spawn operation, while EventNewThread’s getNewThread will return 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.

### EventNewUserThread

An EventNewUserThread triggers when a process spawns a new user-level thread. The Event class’ getThread function will return 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 will return 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.

### EventNewLWP

An EventNewLWP triggers when a process spawns a new LWP. The Event class’ getThread function will return the original Thread that performed the spawn operation, while EventNewThread’s getNewThread will return 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.

### EventThreadDestroy

An EventThreadDestroy triggers when a thread exits. Event’s getThread member function will return 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

### EventUserThreadDestroy

An EventUserThreadDestroy triggers when a thread exits. Event’s getThread member function will return the thread that exited.

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

Associated EventType Code:

UserThreadDestroy

### EventLWPDestroy

An LWPThreadDestroy triggers when a thread exits. Event’s getThread member function will return the thread that exited.

Associated EventType Code:

LWPDestroy

### 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.

### 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.

### 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.

### 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

### 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.

1. 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.

Defined 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 will return 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

ebx

ecx

edx

ebp

esp

esi

edi

oeax

eip

flags

cs

ds

es

fs

gs

ss

fsbase

gsbase

namespace x86\_64

rax

rbx

rcx

rdx

rbp

rsp

rsi

rdi

r8

r9

r10

r11

r12

r13

r14

r15

orax

rip

flags

cs

ds

es

fs

gs

ss

fsbase

gsbase

namespace ppc32

r0

r1

r2

r3

r4

r5

r6

r7

r8

r9

r10

r11

r12

r13

r14

r15

r16

r17

r18

r19

r20

r21

r22

r23

r24

r25

r26

r27

r28

r29

r30

r31

fpscw

lr

cr

xer

ctr

pc

msr

namespace ppc64

r0

r1

r2

r3

r4

r5

r6

r7

r8

r9

r10

r11

r12

r13

r14

r15

r16

r17

r18

r19

r20

r21

r22

r23

r24

r25

r26

r27

r28

r29

r30

r31

fpscw

lr

cr

xer

ctr

pc

msr

addBreakpoint, Process 15

allThreadsRunning, Process 13

allThreadsStopped, Process 13

Architecture 37

attachProcess, Process 9, 11

begin, LibraryPool 26

begin, RegisterPool 27

begin, ThreadPool 24

Breakpoint 15, 16, 20, 33

callback function 2, 4, 6, 11, 28

cb\_func\_t, Process 11

cb\_ret\_t, Process 2, 6, 10

code, EventType 29

const\_iterator, LibraryPool 25

const\_iterator, ThreadPool 24

const\_ptr 8

continueProc, Process 6, 13

continueThread, Thread 6, 17

controller process 1

control-transfer breakpoints 20

createIRPC, IRPC 22

createProcess, Process 9, 10

detach, Process 14

end, LibraryPool 26

end, RegisterPool 27

end, ThreadPool 24

Event 2, 4, 5, 30

EventBreakpoint 33

EventCrash 33

EventExec 33

EventExit 32

EventFork 34

EventIRPC 7

EventLibrary 35

EventNewThread 34

EventNotify 6, 27

EventRPC 35

EventSignal 35

EventSingleStep 35

EventStop 33

EventTerminate 32

EventThreadDestroy 34

EventType 4, 11, 28, 30

EventType constructor 29

EventType, Code 5, 29

EventType, Time 28

eventtype\_cmp 29

evNotify 27

find,ThreadPool 24

freeMemory, Process 14

getAddress, EventBreakpoint 34

getAddress, IRPC 22

getAllRegisters, Thread 18

getArchAddressWidth 37

getArchitecture, MachRegister 38

getArchitecture, Process 12

getBaseRegister, MachRegister 38

getBinaryCodeBlob, IRPC 22

getBinaryCodeSize, IRPC 22

getBreakpoints, EventBreakpoint 34

getChildProcess, EventFork 35

getCrashSignal, Process 13

getData, Breakpoint 21

getDataLoadAddress, Library 20

getEventBreakpoint, Event 31

getEventCrash, Event 31

getEventDestroy, Event 31

getEventExec, Event 31

getEventExit, Event 31

getEventFork, Event 31

getEventLibrary, Event 31

getEventNewThread, Event 31

getEventRPC, Event 31

getEventSignal, Event 31

getEventSingleStep, Event 31

getEventStop, Event 31

getEventTerminate, Event 31

getEventType, Event 31

getExecPath, EventExec 33

getExecutable, LibraryPool 26

getExitCode, EventExit 32

getExitCode, Process 12

getFD, EventNotify 6, 27

getFramePointer, MachRegister 37

getID 22

getInitialThread,ThreadPool 25

getIRPC, EventRPC 35

getLibraryByName, LibraryPool 26

getLoadAddress, Library 20

getLWP, Thread 17

getName, Library 20

getNewThread, EventNewThread 34

getPC, MachRegister 37

getPid, Process 12

getPostedIRPCs, Process 16

getPostedIRPCs, Thread 19

getProcess, Event 30

getProcess, Thread 17

getProcess,ThreadPool 24

getRegister, Thread 17

getRunningIRPC, Thread 19

getSignal, EventSignal 35

getSingleStepMode, Thread 19

getStackPointer, MachRegister 37

getStartOffset, IRPC 23

getSubRegVal, MachRegister 38

getSyncType, Event 31

getTermSignal, EventCrash 33

getThread, Event 30

getToAddress, Breakpoint 21

handleEvents, Process 3, 7, 8, 11, 27

hasRunningThread, Process 13

hasStoppedThread, Process 13

IRPC 7, 16, 21, 35

isCrashed, Process 13

isCtrlTransfer, Breakpoint 21

isExited, Process 12

isFramePointer, MachRegister 38

isInitialThread, Thread 17

isLive, Thread 17

isPC, MachRegister 38

isRunning, Thread 17

isStackPointer, MachRegister 38

isStopped, Thread 17

isTerminated, Process 12

isValid, MachRegister 38

iterator, Library 25

iterator, RegisterPool 26

iterator, ThreadPool 24

libraries, Process 14

Library 19

LibraryPool 14

libsAdded, EventLibrary 36

libsRemoved, EventLibrary 36

load address 19

MachRegister 17, 18, 37

MachRegisterVal 17, 18, 37

mallocMemory, Process 14

name, Event 31

name, EventType 30

name, MachRegister 38

namespaces 9

newBreakpoint, Breakpoint 21

newTransferBreakpoint, Breakpoint 21

notify\_cb\_t, EventNotify 27

postIRPC, Process 7, 16

postIRPC, Thread 7, 18

ppc32 registers 39

ppc64 registers 39

Process 2, 4, 9, 24

ptr 8

readMemory, Process 15

registerCB, EventNotify 28

registerEventCallback, Process 11

RegisterPool 18

removeCB, EventNotify 28

removeEventCallback, Process 12

rmBreakpoint, Process 16

running state 4, 13

setAllRegisters, Thread 18

setData, Breakpoint 21

setRegister, Thread 18

setSingleStep, Thread 35

setSingleStepMode, Thread 19

setStartOffset, IRPC 23

size, LibraryPool 26

size, MachRegister 38

size, RegisterPool 27

size,ThreadPool 24

stopped state 4, 13

stopProc, Process 6, 13

stopThread, Thread 6, 17

SyncType, Event 30

target process 1, 2

terminate, Process 14

Thread 4, 16, 23

ThreadPool 14, 23

threads, Process 14

time, EventType 29

val, MachRegister 38

writeMemory, Process 15

x86 registers 38

x86\_64 registers 39