# 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*. ProcControlAPI allows the controller process to perform operate on target processes, such as writing to a target process’ memory, stopping and running threads in a target process, or receiving notice when certain events occur in the target process. ProcControlAPI presents these operations through a platform-independent API and high-level abstractions. Users can describe what they want ProcControlAPI to do, and let ProcControlAPI handle 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 as the underlying library performing operations such as creating process, running threads, and handling breakpoints in the target processes.

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 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. The user can provides a set of functions to call when certain event types occur 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 these arguments. ProcControlAPI will spawn a new target process and leave it in a stopped state to prevent it from executing yet.

#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 it 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 providing 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 details are presented in Section 2.

## Processes and Threads

## Events and EventType

## Callbacks

## Event Handling

## Memory Management

# API Reference

## Class 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 for 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 will 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\_ret\_t {

cbDefault,

cbThreadContinue,

cbThreadStop,

cbProcContinue,

cbProcStop

}

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 or continued by respectively returning cbThreadContinue, cbThreadStop, cbProcContinue, or cbProcStop. The cbDefault return value has ProcControlAPI make a stop or continue decision based on the type of event that was given to the callback.

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)

This function creates a new process by executing an executable file named by executable with the arguments specified by argv, and 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.

ProcControlAPI may deliver callbacks from 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 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 from 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 occurs requires a callback ProcControlAPI will need control of the main thread in order to deliver a callback. This function gives control of the main thread to ProcControlAPI for it to deliver any callbacks. A user can know when to call handleEvents by using the EventNotify interface; see the section on EventNotify for more details.

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. The Dyninst::Architecture type is defined in dyn\_regs.h and is:

enum Architecture {  
 Arch\_none,  
 Arch\_x86,  
 Arch\_x86\_64,  
 Arch\_ppc32,  
 Arch\_ppc64  
 }

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 of 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 of the error.

bool hasRunningThread() const

This function will return true of 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 of the error.

bool allThreadsStopped() const

This function will return true 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 of the error.

bool allThreadsRunning() const

This function will return true 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 of the error.

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 from 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 from 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 on ThreadPool for more details.

const LibraryPool &libraries() const

ThreadPool &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 on LibraryPool for more details.

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 from 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 from 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 copies to an area of memory from the controller process to the target process. 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 copies an area of memory from the target process to the controller process. 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 on Breakpoint for more details.

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 on Breakpoint for more details.

bool postIRPC(IRPC::ptr irpc) const

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

## Class 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 will be cleaned when there are no more references. ProcControlAPI will maintain internal references to any Thread it actively controls, relinquishing those references when the process either 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.

Process::ptr getProcess() const

This function returns 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 from 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 from 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.

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 sets the value of a single register in this thread to a new value. The register is specified by the reg parameter, and the value that should be written to reg is specified by the val parameter.

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 gets the values of every register in the thread and returns them as part of the RegisterPool pool. Depending on the OS, this call may be more efficient that calling Thread::getRegister multiple times.

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 pool. Depending on the OS, this call may be more efficient that calling Thread::setRegister multiple times.

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 postIRPC(IRPC::ptr irpc) const

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

IRPC::const\_ptr getRunningIRPC() const

## Library

## Breakpoint

## IRPC

## ThreadPool

## LibraryPool

## RegisterPool

## EventNotify

## EventType

## Event

### EventTerminate

### EventExit

### EventCrash

### EventExec

### EventStop

### EventBreakpoint

### EventNewThread

### EventThreadDestroy

### EventFork

### EventSignal

### EventRPC

### EventSingleStep

### EventLibrary

addBreakpoint 11

allThreadsRunning 8

allThreadsStopped 8

attachProcess 5

cb\_func\_t 4

cb\_ret\_t 4

continueProc 9

continueThread 13

createProcess 5

detach 9

freeMemory 10

getAllRegisters 14

getArchitecture 7

getCrashSignal 8

getExitCode 8

getLWP 12

getPid 7

getPostedIRPCs 12, 15

getProcess 12

getRegister 14

getRunningIRPC 15

handleEvents 5

hasRunningThread 8

hasStoppedThread 8

isCrashed 8

isExited 8

isInitialThread 13

isLive 13

isRunning 13

isStopped 13

isTerminated 7

libraries 10

mallocMemory 10

postIRPC 12, 15

Process::const\_ptr 4

Process::ptr 4

readMemory 11

registerEventCallback 6

removeEventCallback 6, 7

rmBreakpoint 12

setAllRegisters 14

setRegister 14

stopProc 9

stopThread 13

terminate 9

Thread::const\_ptr 12

Thread::ptr 12

threads 10

writeMemory 11