Paradyn Parallel Performance Tools

StackwalkerAPI Programmer's Guide

Release 1.0
June 2009

Paradyn Project
www.paradyn.org

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
paradyn@cs.wisc.edu

Computer Sciences Department
University of Maryland
College Park, MD 20742
bugs@dyninst.org

Table Of Contents

1 INTRODUCTION

This document describes StackwalkerAPI, an API and library for walking a call stack. The call stack (also known as the run-time stack) is a stack found in a process that contains the currently active stack frames. Each stack frame is a record of an executing function (or function-like object such as a signal handler or system call). StackwalkerAPI provides an API that allows users to collect a call stack (known as *walking* the call stack) and access information about its stack frames. The current implementation supports Linux/x86, Linux/AMD-64, Linux/Power, and BlueGene/L.

StackwalkerAPI is designed to be both easy-to-use and easy-to-extend. Users can easily use StackwalkerAPI to walk a call stack without needing to understand how call stacks are laid out on their platform. Users can easily extend StackwalkerAPI to work with new platforms and types of stack frames by implementing a set of callbacks that can be plugged into StackwalkerAPI.

StackwalkerAPI's ease-of-use comes from it providing a platform independent interface that allows users to access detailed information about the call stack. For example, the following C++ code-snippet is all that is needed to walk and print the call stack of the currently running thread.

```
std::vector<Frame> stackwalk;
string s;

Walker *walker = Walker::newWalker();
walker->walkStack(stackwalk);
for (unsigned i=0; i<stackwalk.size(); i++) {
      stackwalk[i].getName(s);
      cout << “Found function “ << s << endl;
}
```

StackwalkerAPI can walk a call stack in the same address space as where the StackwalkerAPI library lives (known as a *first party* stackwalk), or it can walk a call stack in another process (known as a *third party* stackwalk). To change the above example to perform a third party stackwalk, we would only need to pass a process identifier to `newWalker`, e.g:

```
Walker *walker = Walker::newWalker(pid);
```

Our other design goal with StackwalkerAPI is to make it easy-to-extend. The mechanics of how to walk through a stack frame can vary between different platforms, and even between different types of stack frames on the same platform. In addition, different platforms may have different mechanisms for reading the data in a call stack or looking up symbolic names that go with a stack frame. StackwalkerAPI provides a callback interface for plugging in mechanisms for handling new systems and types of stack frames. The callback interface can be used to port StackwalkerAPI to new platforms, extend StackwalkerAPI support on existing systems, or more easily integrate StackwalkerAPI into existing tools. There are callbacks for the following StackwalkerAPI operations:

- *Walk through a stack frame* - StackwalkerAPI will find different types of stack frames on different platforms and even within the same platform. For example, on Linux/x86 the stack frame generated by a typical function looks different from the stack frame generated by a signal handler. The callback interface can be used to register a handler with StackwalkerAPI that knows how to walk through a new type of stack frame. For example, the DyninstAPI tool registers an object with StackwalkerAPI that describes how to walk through the stack frames generated by its instrumentation.

- *Access process data* - To walk a call stack, StackwalkerAPI needs to be able to read a process' memory and registers. When doing a first party stackwalk, this is done by directly reading them from the current address space. When doing a third party stackwalk, this is done by reading them using a debugger interface. The callback interface can be used to register new objects for accessing process data. This can be used, for example, to port StackwalkerAPI to a new operating system or make it work with a new debugger interface.
- *Look up symbolic names* - When StackwalkerAPI finds a stack frame, it gets an address that points into the piece of code that created that stack frame. This address is not necessarily meaningful to a user, so StackwalkerAPI attempts to associate the address with a symbolic name. The callback interface can be used to register an object with StackwalkerAPI that performs an address to name mapping, allowing StackwalkerAPI to associate names with stack frames.

2 ABSTRACTIONS

StackwalkerAPI contains two interfaces: the Stackwalking Interface and the Callback Interface. The stackwalking interface is used to walk the call stack, query information about stack frames, and collect basic information about threads. The Callback Interface is used to provide custom mechanisms for walking a call stack. Users who operate in one of StackwalkerAPI's standard configurations do not need to use the Callback Interface.

Figure 1 shows the ownership hierarchy for StackwalkerAPI's classes. Ownership is a "contains" relationship; if one class owns another, then instances of the owner class maintain an exclusive instance of the other. For example, in Figure 1 the each `Walker` instance contains exactly one instance of a `ProcessState` object. No other instance of `Walker` uses that instance of `ProcessState`.

This remainder of this section briefly describes the six classes that make up StackwalkerAPI's two interfaces. For more details, see the class descriptions in Section 3.

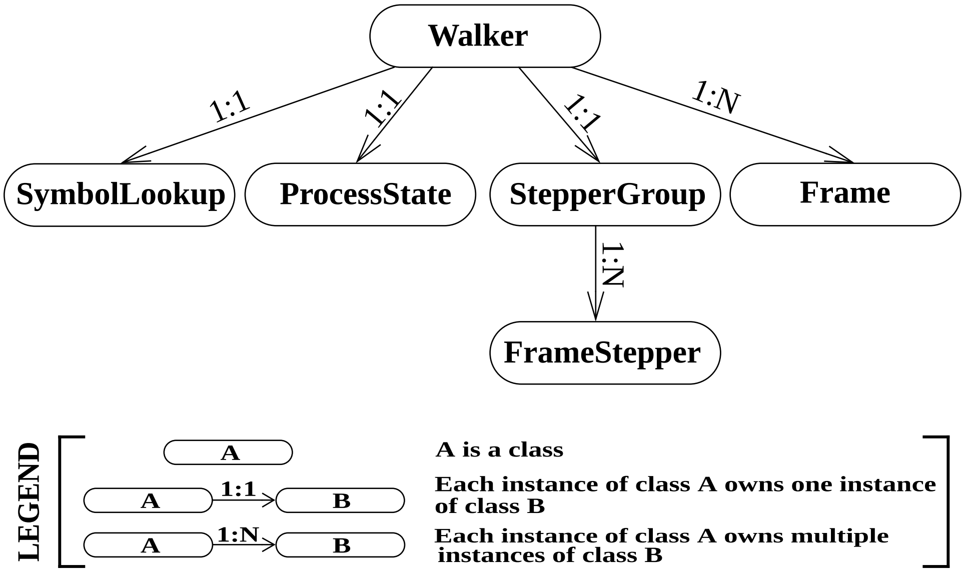

Walker
1:1
1:1
1:1
1:N
SymbolLookup
ProcessState
StepperGroup
Frame
1:N
FrameStepper
LEGEND
A
A is a class
A
1:1
B
Each instance of class A owns one instance of class B
A
1:N
B
Each instance of class A owns multiple instances of class B

Figure 1: Object Ownership

2.1 Stackwalking Interface

- *`Walker`* - The `Walker` class is the top-level class used for collecting stackwalks. It provides a simple interface for requesting a stackwalk. Each `Walker` object is associated with one process, but may walk the call stacks of multiple threads within that process.
- *`Frame`* - A call stack is returned as a vector of `Frame` objects, where each `Frame` object represents a stack frame. It can provide information about the stack frame and basic information about the function, signal handler or other mechanism that created it. Users can request information such as the symbolic name associated with the `Frame` object, and values of its saved registers.

2.2 Callback Interface

StackwalkerAPI includes default implementations of the Callback Interface on each of its supported platforms. These default implementations allow StackwalkerAPI to work "out of the box" in a standard configuration on each platform. Users can port StackwalkerAPI to new platforms or customize its call stack walking behavior by implementing their own versions of the classes in the Callback Interface.

- *`FrameStepper`* - A `FrameStepper` object describes how to walk through a single type of stack frame. Users can provide an implementation of this interface that allows StackwalkerAPI to walk through new types of stack frames. For example, the DyninstAPI uses this interface to extend StackwalkerAPI to allow it to walk through stack frames created by instrumentation code.
- *`StepperGroup`* - A `StepperGroup` is a collection of `FrameStepper` objects and criteria that describes when to use each type of `FrameStepper`. These criteria are based on simple address ranges in the code space of the target process. In the above example with DyninstAPI, it would be the job of the `StepperGroup` to identify a stack frame as belonging to instrumentation code and use the instrumentation `FrameStepper` to walk through it.
- *`ProcessState`* - A `ProcessState` interface describes how to access data in the target process. To walk a call stack, StackwalkerAPI needs to access both registers and memory in the target process; `ProcessState` provides an interface that StackwalkerAPI can use to access that information. StackwalkerAPI includes two default implementation of `ProcessState` for each platform: one to collect a first party stackwalk in the current process, and one that uses a debugger interface to collect a third party stackwalk in another process.
- *`SymbolLookup`* - The `SymbolLookup` interface is used to associate a symbolic name with a stack frame. A stackwalk returns a collection of addresses in the code space of a binary. This class uses the binary's symbol table to map those addresses into symbolic names. A default implementation of this class, which uses the DynSymtab package, is provided with StackwalkerAPI. A user could, for example, use this interface to allow StackwalkerAPI to use libelf to look up symbol names instead.

3 API REFERENCE

This section describes the StackwalkerAPI API. It is divided into three sub-sections: a description of the definitions and basic types used by this API, a description of the interface for collecting stackwalks, and a description of the callback interface.

3.1 Definitions and Basic Types

The following definitions and basic types are referenced throughout the rest of this manual.

3.1.1 Definitions

- *Stack Frame* - A stack frame is a record of a function (or function-like object) invocation. When a function is executed, it may create a frame on the call stack. StackwalkerAPI finds stack frames and returns a description of them when it walks a call stack.

The following three definitions deal with stack frames.

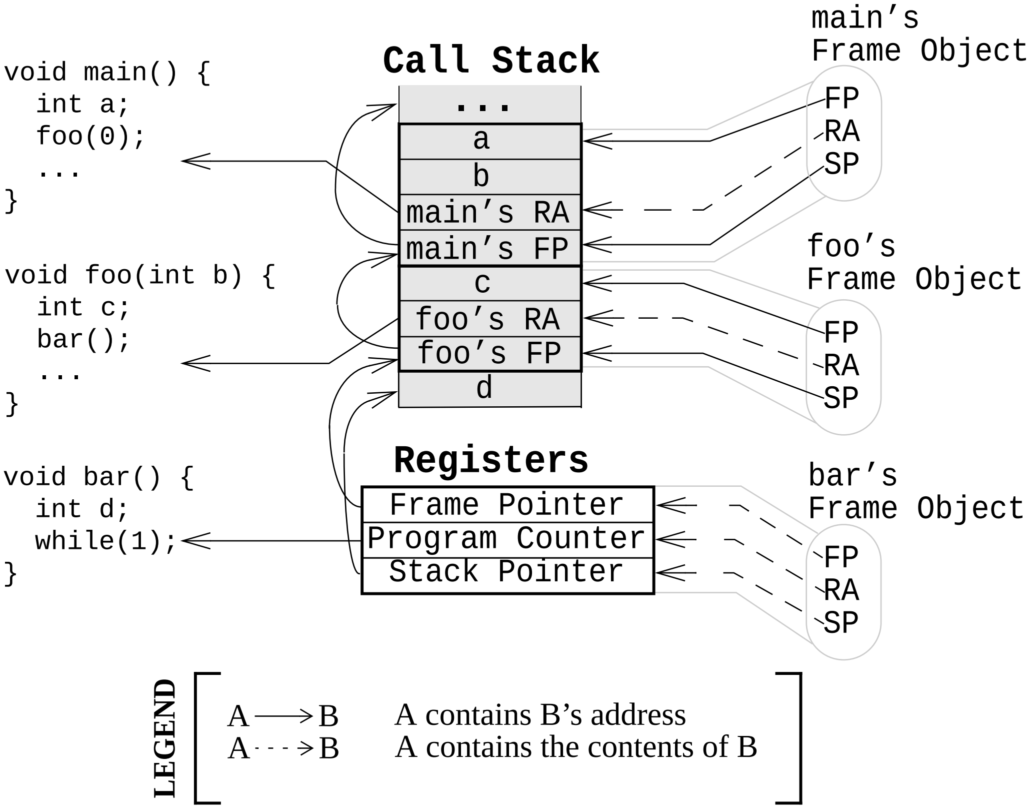
- *Bottom of the Stack* - The bottom of the stack is the earliest stack frame in a call stack, usually a thread’s initial function. The stack grows from bottom to the top.
- *Top of the Stack* - The top of the stack is the most recent stack frame in a call stack. The stack frame at the top of the stack is for the currently executing function.
- *Frame Object* - A `Frame` object is StackwalkerAPI’s representation of a stack frame. A `Frame` object is a snapshot of a stack frame at a specific point in time. Even if a stack frame changes as a process executes, a `Frame` object will remain the same. Each `Frame` object is represented by an instance of the `Frame` class.

The following three definitions deal with fields in a `Frame` object.

- *SP (Stack Pointer)* - A `Frame` object’s SP member points to the top of its stack frame (a stack frame grows from bottom to top, similar to a call stack). The `Frame` object for the top of the stack has a SP that is equal to the value in the stack pointer register at the time the `Frame` object was created. The `Frame` object for any other stack frame has a SP that is equal to the top address in the stack frame.
- *FP (Frame Pointer)* - A `Frame` object’s FP member points to the beginning (or bottom) of its stack frame. The Frame object for the top of the stack has a FP that is equal to the value in the frame pointer register at the time the `Frame` object was created. The `Frame` object for any other stack frame has a FP that is equal to the beginning of the stack frame.
- *RA (Return Address)* - A `Frame` object’s RA member points to the location in the code space where control will resume when the function that created the stack frame resumes. The Frame object for the top of the stack has a RA that is equal to the value in the program counter register at the time the `Frame` object was created. The `Frame` object for any other stack frame has a RA that is found when walking a call stack.

Figure 2 shows the relationship between application code, stack frames, and `Frame` objects. In the figure, the source code on the left has run through the `main` and `foo` functions, and into the `bar` function. It has created the call stack in the center, which is shown as a sequence of words growing down. The current values of the processor registers, while executing in `bar`, are shown below the call stack. When StackwalkerAPI walks the call stack, it creates the `Frame` objects shown on the right. Each `Frame` object corresponds to one of the stack frames found in the call stack or application registers.

The call stack in Figure 2 is similar to one that would be found on the x86 architecture. Details about how the call stack is laid out may be different on other architectures, but the meanings of the FP, SP, and RA fields in the Frame objects will remain the same.



```
void main() {
  int a;
  foo(0);
  ...
}

void foo(int b) {
  int c;
  bar();
  ...
}

void bar() {
  int d;
  while(1);
}
```

Figure 2: Stack Frame and Frame Object Layout

The following four definitions deal with processes involved in StackwalkerAPI.

- *Target Process* - The process from which StackwalkerAPI is collecting stackwalks.
- *Host Process* - The process in which StackwalkerAPI code is currently running.
- *First Party Stackwalk* - StackwalkerAPI collects first party stackwalk when it walks a call stack in the same address space it is running in, i.e. the target process is the same as the host process.
- *Third Party Stackwalk* - StackwalkerAPI collects third party stackwalk when it walks the call stack in a different address space from the one it is running in, i.e. the target process is different from the host process. A third party stackwalk is usually done through a debugger interface.

3.1.2 Basic Types

`typedef unsigned long` **`Address`**

An integer value capable of holding an address in the target process. Address variables should not, and in many cases cannot, be used directly as a pointer. It may refer to an address in a different process, and it may not directly match the target process' pointer representation. `Address` is guaranteed to be at least large enough to hold an address in a target process, but may be larger.

`typedef ...` **`Dyninst::PID`**

A handle for identifying a process. On UNIX systems this will be an integer representing a PID. On Windows this will be a `HANDLE` object.

`typedef ...` **`Dyninst::THR_ID`**

A handle for identifying a thread. On Linux based platforms this is an integer referring to a TID (Thread Identifier). On Solaris and AIX the integer refers to a LWP (Light Weight Process). On Windows it is a `HANDLE` object.

`typedef ...` **`Dyninst::MachRegister`**

A value that names a machine register.

`typedef unsigned long` **`Dyninst::MachRegisterVal`**

A value that holds the contents of a register. A `Dyninst::MachRegister` names a specific register, while a `Dyninst::MachRegisterVal` represents the value that may be in that register.

3.2 Namespace StackwalkerAPI

The classes in Section 3.3 and Section 3.4 fall under the C++ namespace `Dyninst::Stackwalker`. To access them, a user should refer to them using the `Dyninst::Stackwalker::` prefix, e.g. `Dyninst::Stackwalker::Walker`. Alternatively, a user can add the C++ `using` keyword above any references to StackwalkerAPI objects, e.g, `using namespace Dyninst` and `using namespace Stackwalker`.

3.3 Stackwalking Interface

This section describes StackwalkerAPI's interface for walking a call stack. This interface is sufficient for walking call stacks on all the systems and variations covered by our default callbacks.

To collect a stackwalk, first create new Walker object associated with the target process via

```
Walker::newWalker()
```

or

```
Walker::newWalker(Dyninst::PID pid).
```

Once a `Walker` object has been created, a call stack can be walked with the

```
Walker::walkStack
```

method. The new stack walk is returned as a vector of `Frame` objects.

3.3.1 Class Walker

The `Walker` class allows users to walk call stacks and query basic information about threads in a target process. The user should create a `Walker` object for each process from which they are walking call stacks. Each `Walker` object is associated with one process, but may walk call stacks on multiple threads within that process. The `Walker` class allows users to query for the threads available for walking, and it allows you to specify a particular thread whose call stack should be walked. Stackwalks are returned as a vector of `Frame` objects.

Each `Walker` object contains three objects:

- `ProcessState`
- `StepperGroup`
- `SymbolLookup`

These objects are part of the Callback Interface and can be used to customize StackwalkerAPI. The `ProcessState` object tells `Walker` how to access data in the target process, and it determines whether this `Walker` collects first party or third party stackwalks. `Walker` will pick an appropriate default `ProcessState` object based on which factory method[1] the users calls. The `StepperGroup` object is used to customize how the `Walker` steps through stack frames. The `SymbolLookup` object is used to customize how StackwalkerAPI looks up symbolic names of the function or object that created a stack frame.

```
static Walker *newWalker()
```

This factory method creates a new `Walker` object that performs first party stackwalks.

The new `Walker` object uses the default `StepperGroup` and `SymbolLookup` callbacks for the current platform, and it uses the `ProcSelf` callback for its `ProcessState` object. See Section 3.4.1 for more information about defaults in the Callback Interface.

This method returns `NULL` if it was unable to create a new `Walker` object. The new `Walker` object was created with the `new` operator, and should be deallocated with the `delete` operator when it is no longer needed.

```
static Walker *newWalker(Dyninst::PID pid)
```

[1] "Factory method" is a object-oriented design pattern term that describes a method that is responsible for constructing new objects. StackwalkerAPI uses factory methods to allow it to return an error when constructing an object. The `Walker::newWalker` methods are the factory methods for the `Walker` class.

This factory method creates a new Walker object that performs third party stackwalks, on the process identified by pid.

The new Walker object uses the default StepperGroup and SymbolLookup callbacks for the current platform, and it uses the ProcSelf callback for its ProcessState object. See Section 3.4.1 for more information about defaults in the Callback Interface.

This method returns NULL if it was unable to create a new Walker object. The new Walker object was created with the new operator, and should be deallocated with the delete operator when it is no longer needed.

```
static Walker *newWalker(ProcessState *proc,
      StepperGroup *steppergroup = NULL,
      SymbolLookup *lookup = NULL)
```

This factory method creates a new Walker object that walks call stacks on the given proc object. Custom StepperGroup and SymbolLookup can be given with the steppergroup and lookup parameters. If the steppergroup or lookup parameters are NULL, then Stackwalker-API will create a Walker object that uses a default callbacks for the NULL parameter, as described in Section 3.4.1.

It is an error to pass NULL in the proc parameter. proc is used to determine whether to collect first party or third party stackwalks and, in the case of third party stackwalks, identify the target process.

This method returns NULL if there was an error creating the new Walker object. The new Walker object was created with the new operator, and should be deallocated with the delete operator when it is no longer needed.

```
bool walkStack(std::vector<Frame> &stackwalk, Dyninst::THR_ID thread =
NULL_THR_ID)
```

This method walks a call stack in the process associated with this Walker. The call stack is returned as a vector of Frame objects in stackwalk. The top of the stack is returned in index 0 of stackwalk, and the bottom of the stack is returned in index stackwalk.size()-1.

A stackwalk can be taken on a specific thread by passing a value in the thread parameter. If thread has the value NULL_THR_ID, then a default thread will be chosen. When doing a third party stackwalk, the default thread will be the process' initial thread. When doing a first party stackwalk, the default thread will be the thread that called walkStack.

This method returns true on success and false on failure.

```
bool walkStackFromFrame(std::vector<Frame> &stackwalk, const Frame &frame)
```

This method walks a call stack starting from the given stack frame, `frame`. The call stack will be output in the `stackwalk` vector, with `frame` stored in index `0` of `stackwalk` and the bottom of the stack stored in index `stackwalk.size()-1`.

This method returns `true` on success and `false` on failure.

```
bool walkSingleFrame(const Frame &in, Frame &out)
```

This methods walks through single frame, `in`. Parameter `out` will be set to `in`'s caller frame.

This method returns `true` on success and `false` on error.

```
bool getInitialFrame(Frame &frame, Dyninst::THR_ID thread = NULL_THR_ID)
```

This method returns the `Frame` object on the top of the stack in parameter `frame`. Under `walkStack`, `frame` would be the one returned in index 0 of the `stackwalk` vector.

A stack frame can be found on a specific thread by passing a value in the `thread` parameter. If `thread` has the value `NULL_THR_ID`, then a default thread will be chosen. When doing a third party stackwalk, the default thread will be the process' initial thread. When doing a first party stackwalk, the default thread will be the thread that called `getInitialFrame`.

This method returns `true` on success and `false` on failure.

```
bool getAvailableThreads(std::vector<Dyninst::THR_ID> &threads)
```

This method returns a vector of threads in the target process upon which StackwalkerAPI can walk call stacks. The threads are returned in output parameter `threads`. Note that this method may return a subset of the actual threads in the process. For example, when walking call stacks on the current process, it is only legal to walk the call stack on the currently running thread. In this case, `getAvailableThreads` returns a vector containing only the current thread.

This method returns `true` on success and `false` on failure.

```
ProcessState *getProcessState() const
```

This method returns the `ProcessState` object associated with this `Walker`.

```
StepperGroup *getStepperGroup() const
```

This method returns the `StepperGroup` object associated with this `Walker`.

```
SymbolLookup *getSymbolLookup() const
```

This method returns the `SymbolLookup` object associated with this `Walker`.

3.3.2 Class Frame

The Walker class returns a call stack as a vector of Frame objects. As described in Section 3.3.1, each Frame object represents a stack frame, and contains a return address (RA), stack pointer (SP) and frame pointer (FP). For each of these values, optionally, it stores the location where the values were found. Each Frame object may also be augmented with symbol information giving a function name (or a symbolic name, in the case of non-functions) for the object that created the stack frame.

The Frame class provides a set of functions (getRALocation, getSPLocation and getFPLocation) that return the location in the target process' memory or registers where the RA, SP, or FP were found. These functions may be used to modify the stack. For example, the DyninstAPI uses these functions to change return addresses on the stack when it relocates code. The RA, SP, and FP may be found in a register or in a memory address on a call stack.

```
static Frame *newFrame(regvalue_t ra, regvalue_t sp, regvalue_t fp,
      Walker *walker)
```

This method creates a new Frame object and sets the mandatory data members: RA, SP and FP. The new Frame object is associated with the walker.

The optional location fields can be set by the methods below.

The new Frame object is created with the new operator, and the user should be deallocate it with the delete operator when it is no longer needed.

```
regvalue_t getRA() const
```

This method returns this Frame object's return address.

```
void setRA(regvalue_t val)
```

This method sets this Frame object's return address to val.

```
regvalue_t getSP() const
```

This method returns this Frame object's stack pointer.

```
void setSP(regvalue_t val)
```

This method sets this Frame object's stack pointer to val.

```
regvalue_t getFP() const
```

This method returns this Frame object's frame pointer.

```
void setFP(regvalue_t val)
```

This method sets this `Frame` object's frame pointer to `val`.

```
typedef enum { loc_address, loc_register, loc_unknown } storage_t;
typedef struct {
      union {
            address addr;
            Dyninst::MachRegister reg;
      } val;
      storage_t location;
} location_t;
```

The `location_t` structure is used by the `getRALocation`, `getSPLocation`, and `getFPLocation` methods to describe where in the process a `Frame` object's RA, SP, or FP were found. When walking a call stack these values may be found in registers or memory. If they were found in memory, the `location` field of `location_t` will contain `loc_address` and the `addr` field will contain the address where it was found. If they were found in a register the `location` field of `location_t` will contain `loc_register` and the `reg` field will refer to the register where it was found. If this `Frame` object was not created by a stackwalk (using the `newframe` factory method, for example), and has not had a set location method called, then location will contain `loc_unknown`.

```
location_t getRALocation() const
```

This method returns a `location_t` describing where the RA was found.

```
void setRALocation(location_t newval)
```

This method sets the location of where the RA was found to `newval`.

```
location_t getSPLocation() const
```

This method returns a `location_t` describing where the SP was found.

```
void setSPLocation(location_t newval)
```

This method sets the location of where the SP was found to `newval`.

```
location_t getFPLocation() const
```

This method returns a `location_t` describing where the FP was found.

```
void setFPLocation(location_t newval)
```

This method sets the location of where the FP was found to `newval`.

```
bool getName(std::string &str)
```

This method returns a stack frame's symbolic name. Most stack frames are created by functions, or function-like objects such as signal handlers or system calls. This method returns the name of the object that created this stack frame. For stack frames create by functions, this symbolic name will be the function name. A symbolic name may not always be available for all `Frame` objects, such as in cases of stripped binaries or special stack frames types.

The function name is obtained by using this `Frame` object's RA to call the `SymbolLookup` callback. By default StackwalkerAPI will attempt to use the DynSymtab package to look up symbol names in binaries. If DynSymtab is not found, and no alternative `SymbolLookup` object is present, then this method will return an error.

This method returns `true` on success and `false` on error.

```
bool getObject(void* &obj)
```

In addition to returning a symbolic name (see `getName`) the `SymbolLookup` interface allows for an opaque object, a `void*`, to be associated with a `Frame` object. The contents of this `void*` is determined by the `SymbolLookup` implementation. Under the default implementation that uses DynSymtab, the `void*` points to a `Dyn_Symbol` object or `NULL` if no symbol is found.

This method returns `true` on success and `false` on error.

```
bool getStepper(FrameStepper* &stepper) const
```

This method returns the `FrameStepper` object that was used to construct this `Frame` object in the `stepper` output parameter. This method returns `true` on success and `false` on error.

```
bool getLibOffset(std::string &lib, Dyninst::Offset &offset, void* &symtab)
```

This method returns the DSO (a library or executable) and an offset into that DSO that points to the location within that DSO where this frame was created. `lib` is the path to the library that was loaded, and `offset` is the offset into that library. The return value of the `symtab` parameter is dependent on the `SymbolLookup` implementation--by default it will contain a pointer to a `Dyninst::Symtab` object for this DSO. See the SymtabAPI Programmers Guide for more information on using `Dyninst::Symtab` objects.

3.4 Callback Interface

This subsection describes the Callback Interface for StackwalkerAPI. The Callback Interface is primarily used to port StackwalkerAPI to new platforms, extend support for new types of stack frames, or integrate StackwalkerAPI into existing tools.

The classes in this subsection are interfaces, they cannot be instantiated. To create a new implementation of one of these interfaces, create a new class that inherits from the callback class and implement the necessary methods. To use a new `ProcessState`, `StepperGroup`, or `Symbol-`

Lookup class with StackwalkerAPI, create a new instance of the class and register it with a new Walker object using the

```
Walker::newWalker(ProcessState *, StepperGroup *, SymbolLookup *)
```

factory method (see Section 3.3.1). To use a new FrameStepper class with StackwalkerAPI, create a new instance of the class and register it with a StepperGroup using the

```
StepperGroup::addStepper(FrameStepper *)
```

method (see Section 3.4.3).

Some of the classes in the Callback Interface have methods with default implementations. A new class that inherits from a Callback Interface can optionally implement these methods, but it is not required. If a method requires implementation, it is written as a C++ pure virtual method (`virtual funcName() = 0`). A method with a default implementation is written as a C++ virtual method (`virtual funcName()`).

3.4.1 Default Implementations

The classes described in the Callback Interface are C++ abstract classes, or interfaces. They cannot be instantiated. For each of these classes StackwalkerAPI provides one or more default implementations on each platform. These default implementations are classes that inherit from the abstract classes described in the Callback Interface. If a user creates a Walker object without providing their own FrameStepper, ProcessState, and SymbolLookup objects, then StackwalkerAPI will use the default implementations listed in Table 1.

	StepperGroup	ProcessState	SymbolLookup	FrameStepper
Linux/x86 Linux/x86_64	1.AddrRange	1.ProcSelf 2.ProcDebug	1.SwkSymtab	1.FrameFuncStepper 2.SigHandlerStepper 3.DebugStepper 4.StepperWanderer 5.BottomOfStackStepper
Linux/PPC	1.AddrRange	1.ProcSelf 2.ProcDebug	1.SwkSymtab	1.FrameFuncStepper 2.SigHandlerStepper
BlueGene/L	1.AddrRange	1.ProcDebug	1.SwkSymtab	1.FrameFuncStepper

1. Callback Interface Defaults

3.4.2 Class FrameStepper

The FrameStepper class is an interface that tells StackwalkerAPI how to walk through a specific type of stack frame. There may be many different ways of walking through a stack frame on a platform, e.g, on Linux/x86 there are different mechanisms for walking through system calls, signal handlers, regular functions, and frameless functions. A single FrameStepper describes how to walk through one of these types of stack frames.

A user can create their own FrameStepper classes that tell StackwalkerAPI how to walk through new types of stack frames. A new FrameStepper object must be added to a StepperGroup before it can be used.

In addition to walking through individual stack frames, a FrameStepper tells its StepperGroup when it can be used. The FrameStepper registers address ranges that cover objects in the target process' code space (such as functions). These address ranges should contain the objects that will create stack frames through which the FrameStepper can walk. If multiple FrameStepper objects have overlapping address ranges, then a priority value is used to determine which FrameStepper should be attempted first.

FrameStepper is an interface class; it cannot be instantiated. Users who want to develop new FrameStepper objects should inherit from this class and implement the below virtual functions.

```
typedef enum { gcf_success,
               gcf_stackbottom,
               gcf_not_me,
               gcf_error } gcframe_ret_t

virtual gcframe_ret_t getCallerFrame(const Frame &in, Frame &out) = 0
```

This method walks through a single stack frame and generates a Frame object that represents the caller's stack frame. Parameter in will be a Frame object that this FrameStepper is capable of walking through. Parameter out is an output parameter that this method should set to the Frame object that called in.

There may be multiple ways of walking through a different types of stack frames. Each FrameStepper class should be able to walk through a type of stack frame. For example, on x86 one FrameStepper could be used to walk through stack frames generated by ABI-compliant functions; out's FP and RA are found by reading from in's FP, and out's SP is set to the word below in's FP. A different FrameStepper might be used to walk through stack frames created by functions that have optimized away their FP. In this case, in may have a FP that does not point out's FP and RA. The FrameStepper will need to use other mechanisms to discover out's FP or RA; perhaps the FrameStepper searches through the stack for the RA or performs analysis on the function that created the stack frame.

If getCallerFrame successfully walks through in, it is required to set the following parameters in out. See Section 3.3.2 for more details on the values that can be set in a Frame object:

- *Return Address (RA)* - The RA should be set with the Frame::setRA method.
- *Stack Pointer (SP)* - The SP should be set with the Frame::setSP method.
- *Frame Pointer (FP)* - The FP should be set with the Frame::setFP method

Optionally, getCallerFrame can also set any of following parameters in out:

- *Return Address Location (RALocation)* - The RALocation should be set with the Frame::setRALocation() method.
- *Stack Pointer Location (SPLocation)* - The SPLocation should be set with the

`Frame::setRALocation()` method.

- *Frame Pointer Location (FPLocation)* - The FPLocation should be set with the `Frame::setFPLocation()` method.

  If a location field in `out` is not set, then the appropriate `Frame::getRALocation`, `Frame::getSPLocation` or `Frame::getFPLocation` method will return `loc_unknown`.

  `getCallerFrame` should return `gcf_success` if it successfully walks through `in` and creates an `out` `Frame` object. It should return `gcf_stackbottom` if `in` is the bottom of the stack and there are no stack frames below it. It should return `gcf_not_me` if `in` is not the correct type of stack frame for this `FrameStepper` to walk through. StackwalkerAPI will then attempt to locate another `FrameStepper` to handle `in` or abort the stackwalk. It should return `gcf_error` if there was an error and the stack walk should be aborted.

```
virtual void registerStepperGroup(StepperGroup &steppergroup)
```

This method is used to notify a `FrameStepper` when StackwalkerAPI adds it to a `StepperGroup`. The `StepperGroup` to which this `FrameStepper` is being added is passed in parameter `steppergroup`. This method can be used to initialize the `FrameStepper` (in addition to any `FrameStepper` constructor).

```
virtual unsigned getPriority() = 0
```

This method is used by the `StepperGroup` to decide which `FrameStepper` to use if multiple `FrameStepper` objects are registered over the same address range (see `addAddressRanges` in Section 3.4.3 for more information about address ranges). This method returns an integer representing a priority level, the lower the number the higher the priority.

The default `FrameStepper` objects provided by StackwalkerAPI all return priorities between `0x1000` and `0x2000`. If two `FrameStepper` objects have an overlapping address range, and they have the same priority, then the order in which they are used is undefined.

3.4.3 Class StepperGroup

The `StepperGroup` class contains a collection of `FrameStepper` objects. The `StepperGroup`'s primary job is to decide which `FrameStepper` should be used to walk through a stack frame given a return address. The default `StepperGroup` keeps a set of address ranges for each `FrameStepper`. If multiple `FrameStepper` objects overlap an address, then the default `StepperGroup` will use a priority system to decide.

`StepperGroup` provides both an interface and a default implementation of that interface. Users who want to customize the `StepperGroup` should inherit from this class and re-implement any of the below virtual functions.

```
virtual StepperGroup(Walker *walker)
```

This factory constructor creates a new `StepperGroup` object associated with `walker`.

```
virtual bool addStepper(FrameStepper *stepper)
```

This method adds a new `FrameStepper` to this `StepperGroup`. The newly added `stepper` will be tracked by this `StepperGroup`, and it will be considered for use when walking through stack frames.

This method returns true if it successfully added the `FrameStepper`, and false on error.

```
virtual bool addAddressRanges(
      const std::vector<std::pair<Address, Address> >&ranges,
      const FrameStepper *stepper) = 0
```

This method associates a set of address ranges, `ranges`, with a `FrameStepper`, `stepper`. These address ranges contain objects in the process' code space that create stack frames that `stepper` can walk through.

The default `StepperGroup` will use `stepper` to walk through a `Frame` object (by returning it from `findStepperForAddr`) if the `Frame` object's RA falls within a range registered by this method. A `Frame` object, `frame`, falls within a range, `range[i]`, if `range[i].first` ≤ `frame.getRA()` < `range[i].second`. If multiple `FrameStepper` objects have overlapping ranges, then the default `StepperGroup` will use the one with the highest priority first (see `FrameStepper::getPriority` in Section 3.4.2).

For example, suppose this `FrameStepper` was designed to walk through a signal handler frame on Linux/x86. During initialization the `FrameStepper` inspects the target process' vsyscall page[2] and finds that signal handlers will appear on the call stack with a RA between `0xffffe000` and `0xffffe400`. It then registers this range with its `StepperGroup` using `addAddressRanges`. If the `StepperGroup` encounters an RA in this range, it then uses the signal handler `FrameStepper` to walk through it.

Suppose another `FrameStepper` was designed to walk through regular stack frames created by ABI-compliant functions. This `FrameStepper` will be used as a general catch-all if no other `FrameStepper` can walk through a `Frame` object. The `FrameStepper` can register itself with an address range that spans the whole address space, and a lower priority than the signal handler `FrameStepper`. The `StepperGroup` will then use the signal handler `FrameStepper` to step through signal handlers, and this `FrameStepper` to step through any other `Frame` object.

[2] The vsyscall page is a small shared object that is loaded by the kernel into every process' address space. It is part of Linux's mechanism for quickly transferring control between the kernel and user space. It also provides information about how to stack walk through system calls and signal handlers.

This method returns `true` on success and `false` if there is an error.

```
virtual bool removeAddressRanges(
      const std::vector<std::pair<Address, Address > > &ranges,
      const FrameStepper *stepper) = 0
```

This method removes a `FrameStepper`'s address range from a `StepperGroup`. See `addAddressRange` for more details on how `StepperGroup` and `FrameStepper` objects use address ranges. The address ranges specified by `ranges` will be deleted from `stepper`'s address ranges. For example, if the address range `0x1000` to `0x2000` was registered to a `FrameStepper` named `foo`, and then `removeAddressRanges` was used to remove the address range `0x1500` to `0x1600` out of `foo`, then `foo` would have two address ranges associated with it: `0x1000` to `0x1500` and `0x1600` to `0x2000`.

This function returns true on success and false on error.

```
virtual bool findStepperForAddr(Address addr, FrameStepper* &out,
      const FrameStepper *last_tried = NULL)
```

Given an address that points into a function (or function-like object), `addr`, this method decides which `FrameStepper` should be used to walk through the stack frame created by the function at that address. A pointer to the `FrameStepper` will be returned in parameter `out`.

It may be possible that the `FrameStepper` this method decides on is unable to walk through the stack frame (it returns `gcf_not_me` from `FrameStepper::getCallerFrame`). In this case StackwalkerAPI will call `findStepperForAddr` again with the `last_tried` parameter set to the failed `FrameStepper`. `findStepperForAddr` should then find another `FrameStepper` to use. Parameter `last_tried` will be set to `NULL` the first time `getStepperToUse` is called for a stack frame.

The default version of this method uses address ranges to decide which `FrameStepper` to use. The address ranges are contained within the process' code space, and map a piece of the code space to a FrameStepper that can walk through stack frames created in that code range. If multiple `FrameStepper` objects share the same range, then the one with the highest priority will be tried first.

This method returns `true` on success and `false` on failure.

```
Walker *getWalker() const
```

This method returns the `Walker` object that associated with this `StepperGroup`.

3.4.4 Class ProcessState

The `ProcessState` class is a virtual class that defines an interface through which StackwalkerAPI can access the target process. It allows access to registers and memory, and provides basic information about the threads in the target process. StackwalkerAPI provides two default types of

ProcessState objects: ProcSelf does a first party stackwalk, and ProcDebug does a third party stackwalk.

A new ProcessState class can be created by inheriting from this class and implementing the following functions.

```
virtual bool getRegValue(Dyninst::MachRegister reg, Dyninst::THR_ID thread,
Dyninst::MachRegisterVal &val) = 0
```

This method takes a register name as input, reg, and returns the value in that register in val in the thread thread.

This method returns true on success and false on error.

```
virtual bool readMem(void *dest, Address source, size_t size) = 0
```

This method reads memory from the target process. Parameter dest should point to an allocated buffer of memory at least size bytes in the host process. Parameter source should contain an address in the target process to be read from. If this method succeeds, size bytes of memory is copied from source, stored in dest, and true is returned. This method returns false otherwise.

```
virtual bool getThreadIds(std::vector<Dyninst::THR_ID> &threads) = 0
```

This method returns a list of threads whose call stacks can be walked in the target process. Thread are returned in the threads vector. In some cases, such as with the default ProcDebug, this method returns all of the threads in the target process. In other cases, such as with ProcSelf, this method returns only the calling thread.

The first thread in the threads vector (index 0) will be used as the default thread if the user requests a stackwalk without specifying an thread (see Walker::WalkStack)

This method returns true on success and false if an error occurs.

```
virtual bool getDefaultThread(Dyninst::THR_ID &default_tid) = 0
```

This method returns the thread representing the initial process in the default_tid output parameter.

This method returns true on success and false if an error occurs.

```
virtual Dyninst::PID getProcessID() = 0
```

This method returns a process ID for the target process. The default ProcessState implementations (ProcDebug and ProcSelf) will return a PID on UNIX systems and a HANDLE object on Windows

3.4.4.1 Class LibraryState

`LibraryState` is a helper class for `ProcessState` that provides information about the current DSOs (libraries and executables) that are loaded into a process' address space. `FrameSteppers` frequently use the `LibraryState` to get the DSO through which they are attempting to stack walk.

Each Library is represented using a `LibAddrPair` object, which is defined as follows:

```
typedef std::pair<std::string, Dyninst::Address> LibAddrPair
```

`LibAddrPair.first` refers to the file path of the library that was loaded, and `LibAddrPair.second` is the load address of that library in the process' address space. The load address of a library can be added to a symbol offset from the file in order to get the absolute address of a symbol.

```
virtual bool getLibraryAtAddr(Address addr, LibAddrPair &lib) = 0
```

This method returns a DSO, using the `lib` output parameter, that is loaded over address `addr` in the current process.

This method returns false if no library is loaded over `addr` or an error occurs, and true if it successfully found a library.

```
virtual bool getLibraries(std::vector<LibAddrPair> &libs) = 0
```

This method returns all DSOs that are loaded into the process' address space in the output vector parameter, `libs`.

This method returns `true` on success and `false` on error.

```
virtual void notifyOfUpdate() = 0
```

This method is called by the ProcessState when it detects a change in the process' list of loaded libraries. Implementations of `LibraryStates` should use this method to refresh their lists of loaded libraries.

```
virtual Address getLibTrapAddress() = 0
```

Some platforms that implement the System/V standard (Linux, Solaris, BlueGene/P) use a trap event to determine when a process loads a library. A trap instruction is inserted into a certain address, and that trap will execute whenever the list of loaded libraries change.

On System/V supported platforms this method should return the address where a trap should be inserted to watch for libraries loading and unloading. The ProcessState object will insert a trap at this address and then call `notifyOfUpdate` when that trap triggers.

On non-System/V platforms this method should return 0.

3.4.5 Class SymbolLookup

The `SymbolLookup` virtual class is an interface for associating a symbolic name with a stack frame. Each `Frame` object contains an address (the RA) pointing into the function (or function-like object) that created its stack frame. However, users do not always want to deal with addresses when symbolic names are more convenient. This class is an interface for mapping a `Frame` object's RA into a name.

In addition to getting a name, this class can also associate an opaque object (via a `void*`) with a `Frame` object. It is up to the `SymbolLookup` implementation what to return in this opaque object.

The default implementation of `SymbolLookup` provided by StackwalkerAPI uses the DynSymtab tool to lookup symbol names. It returns a `Symbol` object in the anonymous `void*`.

```
virtual bool lookupAtAddr(Address addr, string &out_name, void* &out_value) = 0
```

This method takes an address, `addr`, as input and returns the function name, `out_name`, and an opaque value, `out_value`, at that address. Output parameter `out_name` should be the name of the function that contains `addr`. Output parameter `out_value` can be any opaque value determined by the `SymbolLookup` implementation. The values returned are used by the `Frame::getName()` and `Frame::getObject()` functions.

This method returns `true` on success and `false` on error.

```
virtual Walker *getWalker()
```

This method returns the `Walker` object associated with this `SymbolLookup`.

```
virtual ProcessState *getProcessSate()
```

This method returns the `ProcessState` object associated with this `SymbolLookup`.

4 CALLBACK INTERFACE DEFAULT IMPLEMENTATIONS

StackwalkerAPI provides one or more default implementations of each of the callback classes described in Section 3.4. These implementations are used by a default configuration of StackwalkerAPI.

4.1 Debugger Interface

This section describes how to use StackwalkerAPI for collecting 3rd party stack walks. In 3rd party mode StackwalkerAPI uses the OS's debugger interface to connect to another process and walk its call stacks. As part of being a debugger StackwalkerAPI receives and needs to handle debug events. When a debugger event occurs, StackwalkerAPI must get control host process in order to receive the debugger event and continue the target process.

To illustrate the complexities with running in 3rd party mode, consider the follow code snippet that uses StackwalkerAPI to collect a stack walk every five seconds.

```
Walker *walker = Walker::newWalker(pid);
std::vector<Frame> swalk;
for (;;) {
      walker->walkStack(swalk);
      sleep(5);
}
```

StackwalkerAPI is running in 3rd party mode, since it attached to the target process, `pid`. As the target process runs it may be generating debug events such a thread creation and destruction, library loads and unloads, signals, forking/execing, etc. When one of these debugger events is generated the OS will pause the target process and send a notice to the host process. The target process will remain paused until the host process handles the debug event and resumes the target process.

In the above example the host process is spending almost all of its time in the `sleep` call. If a debugger event happens during the sleep, then StackwalkerAPI will not be able to get control of the host process and handle the event for up to five seconds. This will cause long pauses in the target process and lead to a potentially very large slowdown.

To work around this problem StackwalkerAPI provides a notification file descriptor. This file descriptor represents a connection between the StackwalkerAPI library and user code. StackwalkerAPI will write a single byte to this file descriptor when a debug event occurs, thus notifying the user code that it needs to let StackwalkerAPI receive and handle debug events. The user code can use system calls such as `select` to watch for events on the notification file descriptor.

The following example illustrates how to properly use StackwalkerAPI to collect a stack walk from another process at a five second interval. Details on the `ProcDebug` class, `getNotificationFD` method, and `handleDebugEvent` method can be found in Section 4.1.1. See the UNIX man pages for more information on the `select` system call. Note that this example does not include all of the proper error handling and includes that should be present when using `select`.

```
Walker *walker = Walker::newWalker(pid);
ProcDebug *debugger = (ProcDebug *) walker->getProcessState();
std::vector<Frame> swalk;

for (;;) {
      walker->walkStack(swalk);

      struct timeval timeout;
      timeout.tv_sec = 5;
      timeout.tv_usec = 0;

      int max = 1;
      fd_set readfds, writefds, exceptfds;
      FD_ZERO(&readfds); FD_ZERO(&writefds); FD_ZERO(exceptfds);
      FD_SET(ProcDebug::getNotificationFD(), &readfds);

      for (;;) {
            int result = select(max, &readfds, &writefds, &exceptfds,
                                &timeout);
            if (FD_ISSET(ProcDebug::getNotificationFD(), readfds)) {
                  //Debug event
                  ProcDebug::handleDebugEvent();
            }

            if (result == 0) {
                  //Timeout
                  break;
            }
      }
}
```

4.1.1 Class ProcDebug

Access to StackwalkerAPI's debugger is through the `ProcDebug` class, which inherits from the `ProcessState` interface. The easiest way to get at a `ProcDebug` object is to cast the return value of `Walker::getProcessState()` into a `ProcDebug`. C++'s `dynamic_cast` operation can be used to test if a `Walker` uses the `ProcDebug` interface:

```
ProcDebug *debugger;
debugger = dynamic_cast<ProcDebug*>(walker>getProcessState());
if (debugger != NULL) {
      //3rd party
      ...
} else {
      //1st party
      ...
}
```

In addition to the handling of debug events, described in Section 4.1, the ProcDebug class provides a process control interface; users can pause and resume process or threads, detach from a process, and test for events such as process death. As an implementation of the `ProcessState` class, `ProcDebug` also provides all of the functionality described in Section 3.4.4.

```
virtual bool pause(Dyninst::THR_ID tid = NULL_THR_ID)
```

This method pauses a process or thread. The paused object will not resume execution until `ProcDebug::resume` is called. If the `tid` parameter is not `NULL_THR_ID` then StackwalkerAPI will pause the thread specified by `tid`. If `tid` is `NULL_THR_ID` then StackwalkerAPI will pause every thread in the process.

When StackwalkerAPI collects a call stack from a running thread it first pauses the thread, collects the stack walk, and then resumes the thread. When collecting a call stack from a paused thread StackwalkerAPI will collect the stack walk and leave the thread paused. This method is thus useful for pausing threads before stack walks if the user needs to keep the returned stack walk synchronized with the current state of the thread.

This method returns `true` if successful and `false` on error.

```
virtual bool resume(Dyninst::THR_ID tid = NULL_THR_ID)
```

This method resumes execution on a paused process or thread. This method only resumes threads that were paused by the `ProcDebug::pause` call, using it on other threads is an error. If the `tid` parameter is not `NULL_THR_ID` then StackwalkerAPI will resume the thread specified by `tid`. If tid is `NULL_THR_ID` then StackwalkerAPI will resume all paused threads in the process.

This method returns `true` if successful and `false` on error.

```
virtual bool detach(bool leave_stopped = false)
```

This method detaches StackwalkerAPI from the target process. StackwalkerAPI will no longer receive debug events on this target process and will no longer be able to collect call stacks from it. This method invalidates the associated `Walker` and `ProcState` objects, they should be cleaned using C++'s `delete` operator after making this call. It is an error to attempt to do operations on these objects after a detach, and undefined behavior may result.

If the `leave_stopped` parameter is `true` StackwalkerAPI will detach from the process but leave it in a paused state so that it does resume progress. This is useful for attaching another debugger back to the process for further analysis. The `leave_stopped` parameter is not supported on the Linux platform and its value will have no affect on the `detach` call.

This method returns `true` if successful and `false` on error.

```
virtual bool isTerminated()
```

This method returns true if the associated target process has terminated and false otherwise. A target process may terminate itself by calling exit, returning from main, or receiving an unhandled signal. Attempting to collect stack walks or perform other operations on a terminated process is illegal an will lead to undefined behavior.

A process termination will also be signaled through the notfication FD. Users should check processes for the `isTerminated` state after returning from `handleDebugEvent`.

```
static int getNotificationFD()
```

This method returns StackwalkerAPI's notification FD. The notification FD is a file descriptor that StackwalkerAPI will write a byte to whenever a debug event occurs that need. If the user code sees a byte on this file descriptor it should call `handleDebugEvent` to let StackwalkerAPI handle the debug event. Example code using getNotificationFD can be found in Section 4.1.

If StackwalkerAPI will only create one notification FD, even if it is attached to multiple 3rd party target processes.

```
static bool handleDebugEvent(bool block = false)
```

When this method is called StackwalkerAPI will receive and handle all pending debug events from each 3rd party target process to which it is attached. After handling debug events each target process will be continued (unless it was explicitly stopped by the `ProcDebug::pause` method) and any bytes on the notification FD will be cleared. It is generally expected that users will call this method when a event is sent to the notification FD, although it can be legally called at any time.

If the `block` parameter is true, then `handleDebugEvents` will block until it has handled at least one debug event. If the block parameter is `false`, then `handleDebugEvents` will handle any currently pending debug events or immediately return if none are available.

StackwalkerAPI may receive process exit events for target processes while handling debug events. The user should check for any exited processes by calling `ProcDebug::isTerminated` after handling debug events.

This method returns `true` if it handled at least one debug events and `false` otherwise.

4.2 FrameSteppers

StackwalkerAPI ships with numerous default implementations of the `FrameStepper` class. Each of these `FrameStepper` implementations allow StackwalkerAPI to walk a type of call frames. Section 3.4.1 describes which `FrameStepper` implementations are available on which platforms. This sections gives a brief description of what each `FrameStepper` implementation does. Each of the following classes implements the `FrameStepper` interface described in Section 3.4.2, so we do not repeat the API description for the classes here.

Several of the FrameSteppers use helper classes (see FrameFuncStepper as an example). Users can further customize the behavior of a FrameStepper by providing their own implementation of these helper classes.

4.2.1 Class FrameFuncStepper

This class implements stack walking through a call frame that is setup with the architectures standard stack frame. For example, on x86 this FrameStepper will be used to walk through stack frames that are setup with a push %ebp/mov %esp,%ebp prolog.

4.2.1.1 Class FrameFuncHelper

FrameFuncStepper uses a helper class, FrameFuncHelper, to get information on what kind of stack frame it's walking through. The FrameFuncHelper will generally use techniques such as binary analysis to determine what type of stack frame the FrameFuncStepper is walking through. Users can have StackwalkerAPI use their own binary analysis mechanisms by providing an implementation of this FrameFuncHelper.

There are two important types used by FrameFuncHelper and one important function:

```
typedef enum {
      unknown_t=0,
      no_frame,
      standard_frame,
      savefp_only_frame,
} frame_type;
```

The frame_type describes what kind of stack frame a function uses. If it does not set up a stack frame then frame_type should be no_frame. If it sets up a standard frame then frame_type should be standard_frame. The savefp_only_frame value currently only has meaning on the x86 family of systems, and means that a function saves the old frame pointer, but does not setup a new frame pointer (it has a push %ebp instruction, but no mov %esp,%ebp). If the FrameFuncHelper cannot determine the frame_type, then it should be assigned the value unknown_t.

```
typedef enum {
      unknown_s=0,
      unset_frame,
      halfset_frame,
      set_frame
} frame_state;
```

The frame_state type determines the current state of function with a stack frame at some point of execution. For example, a function may set up a standard stack frame and have a frame_type of standard_frame, but execution may be at the first instruction in the function and the frame is not yet setup, in which case the frame_state will be unset_frame.

If the function sets up a standard stack frame and the execution point is someplace where the frame is completely setup, then the frame_state should be set_frame. If the function sets up a standard frame and the execution point is at a point where the frame does not yet exist or has been torn down, then frame_state should be unset_frame. The halfset_frame value of frame_state is currently only meaningful on the x86 family of architecture, and should if the function has saved the old frame pointer, but not yet set up a new frame pointer.

```
typedef std::pair<frame_type, frame_state> alloc_frame_t;
virtual alloc_frame_t allocatesFrame(Address addr) = 0;
```

The allocatesFrame function of FrameFuncHelper returns a alloc_frame_t that describes the frame_type of the function at addr and the frame_state of the function when execution reached addr.

If addr is invalid or an error occurs, allocatedFrame should return alloc_frame_t(unknown_t, unknown_s).

4.2.2 Class DebugStepper

This class uses debug information found in a binary to walk through a stack frame. It depends on SymtabAPI to read debug information from a binary, then uses that debug information to walk through a call frame.

Most binaries must be built with debug information (-g with gcc) in order to include debug information that this FrameStepper uses. Some languages, such as C++, automatically include stackwalking debug information for use by exceptions. The DebugStepper class will also make use of this kind of exception stacking information if it is available.

4.2.3 Class SigHandlerStepper

The SigHandlerStepper is used to walk through UNIX signal handlers as found on the call stack. On some systems a signal handler generates a special kind of stack frame that cannot be walked through using normal stack walking techniques.

4.2.4 Class BottomOfStackStepper

The BottomOfStackStepper doesn’t actually walk through any type of call frame. Instead it attempts to detect whether the bottom of the call stack has been reached. If so, BottomOfStackStepper will report gcf_stackbottom from its getCallerFrame method. Otherwise it will report gcf_not_me. BottomOfStackStepper runs with a higher priority than any other FrameStepper class.