```
#include <cxxabi.h>
#include <dlfcn.h>
#include <execinfo.h>

#include <string>
#include<sstream>
#include <thread>
#include <memory>
#include <mutex>
#include <string>
#include "CPPTools/HandlerInfo.h"

#ifndef  CPP_TOOLS_BACKTRACER_H_
#define CPP_TOOLS_BACKTRACER_H_
/*
 LDFLAGS += -rdynamic  should be used to have symbol info
 */
namespace cpp_tools
{

class BackTracer{

  enum { MAX_DEPTH = 256 };
public:

  std::string GetTrace()
  {
      std::string out("Thread id=");
      std::thread::id id = std::this_thread::get_id();
      std::ostringstream os;
      os<< std::hex<<id<<std::flush;
      out += os.str();  out += "\n";

     void *trace[MAX_DEPTH] = {NULL};

     std::lock_guard<std::mutex> l(m_mutex);
     int trace_size = backtrace(trace, MAX_DEPTH);

      using namespace abi;

     for (int i=0; i<trace_size; ++i) {

        Dl_info dlinfo = {NULL, NULL, NULL};
        if(!dladdr(trace[i], &dlinfo)){
           continue;
        }
        const char *symname = dlinfo.dli_sname;
        if(!symname){
            continue;
        }
        int status = -2;
        char *demangled = __cxa_demangle(symname, NULL, 0, &status);
        if( (status == 0) && demangled)
        {
               symname = demangled;
               out += dlinfo.dli_fname;  out += ", ";
               out += symname;  out += "\n";
        }

        if (demangled)
           free(demangled);
     }
     return out;
   }
private:
   mutable std::mutex m_mutex;// "trace functions seems to be not thread safe.

};

using BackTracerPtr = std::shared_ptr<BackTracer>;

   inline std::string GetBackTraceStr()
  {
     cpp_tools::BackTracer tracer;
     std::string trace = tracer.GetTrace();
     return trace;
  }

inline std::string GetCurrentExceptionStr()
  {
   std::exception_ptr pException = std::current_exception();
   if(!pException)
   {
      return "";
   }
   try{
       std::rethrow_exception (pException);
   }catch(std::exception &ex)
   {
      return ex.what();
   }
   catch(...)
   {
       return "Unknown exception";
   }
   return "Coding error in GetCurrentExceptionStr";
  }

 inline std::string GetCurrentExceptionStr(std::exception &ex)
  {
   std::string out = ex.what();
   return out;
  }

 inline void ExceptionLog(const HandlerInfo& info, std::exception* pEx = nullptr)
 {
     std::string msg = info.GetInfo();
     const TraceLevel  traceLevel = info.GetTraceLevel();
     if( (traceLevel == TraceLevel::EXCEPTION_TRACE) )
     {
        if(pEx)
        {
             msg += "Exception trace \n";  msg += GetCurrentExceptionStr(*pEx); msg +="\n";
        }
        else
        {
             msg += "Exception trace \n";  msg += GetCurrentExceptionStr(); msg +="\n";
        }
     }
     else  if( (traceLevel == TraceLevel::BACK_TRACE) )
     {
        msg += "Stack trace:";  msg += GetBackTraceStr(); msg +="\n";
     }
     else  if( (traceLevel == TraceLevel::MAX_TRACE) )
     {
         if(pEx)
        {
             msg += "Exception trace \n";  msg += GetCurrentExceptionStr(*pEx); msg +="\n";
        }
        else
        {
             msg += "Exception trace \n";  msg += GetCurrentExceptionStr(); msg +="\n";
        }
        msg += "Stack trace:";  msg += GetBackTraceStr(); msg +="\n";
     }

     if(!msg.empty())
     {
        Logger loger = info.GetLogger();
        loger(msg);
     }
 }


 inline void ExceptionHandler(const HandlerInfo& info, std::exception* pEx = nullptr)
  {
     ExceptionLog(info, pEx );

     if(info.GetPostAction() == PostAction::STD_DEFAULT )
     {
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        std::abort();
     }
  }




}
```

```
/////////////////////

#include <string>
#include <signal.h>
#include <exception>
#include <map>

#include "CPPTools/BackTracer.h"

#ifndef CPP_TOOLS_SIGNAL_H_
#define CPP_TOOLS_SIGNAL_H_

namespace cpp_tools
{

using OnSignal =  std::function<void(int)>;

namespace signal_impl
{

enum class SynchronizationType{UKNOWN, SYNC, ASYNC };

template <sig_atomic_t signal> struct SignalType
{
   enum {m_signal = signal };
   static std::string ToString(){ return std::to_string(m_signal); }
   static SynchronizationType SyncType(){ return SynchronizationType::UKNOWN; }
 };

template <> std::string  SignalType<SIGFPE>::ToString(){ return "SIGFPE";   }
template <> std::string  SignalType<SIGSEGV>::ToString(){ return "SIGSEGV";}

template <> SynchronizationType  SignalType<SIGFPE>::SyncType(){ return
SynchronizationType::SYNC;  }
template <> SynchronizationType  SignalType<SIGSEGV>::SyncType(){ return
SynchronizationType::SYNC;  }


template <sig_atomic_t signal> struct Handler
{
   static_assert( ((signal <= SIGUNUSED) &&  (signal > 0 )),"This signal has not been
   implemented");
   using Type = SignalType<signal>;
   static Handler& Instance(HandlerInfo& info)
   {
      static Handler handler(info);
      return handler;
   }
  void Handle()
  {
    ExceptionHandler(m_info);
  }
  void  SetHandlerInfo(HandlerInfo& info)
  {
     m_info = info;
  }
   private:
      Handler (HandlerInfo& info):m_info(info)
      {}
   private:
      HandlerInfo                m_info;
};


struct Handlers
{
   static Handlers& Instance()
   {
      static Handlers handlers;
      return handlers;
   }
   template <sig_atomic_t signal> void Install(HandlerInfo& info)
   {
      std::lock_guard<std::mutex> l(m_mutex);
      auto iter= m_signals.find(signal);
      if(iter == m_signals.end())
      {
        m_signals.insert(std::map<sig_atomic_t, void*>::value_type (signal,
        &(Handler<signal>::Instance(info) )   ) );
      }else
      {
         ((Handler<signal> *)(iter->second))->SetHandlerInfo(info);
      }
   }

   template <sig_atomic_t signal> bool Handle()
   {
      std::lock_guard<std::mutex> l(m_mutex);
      auto iter= m_signals.find(signal);
      if(iter != m_signals.end())
      {
         ((Handler<signal> *)(iter->second))->Handle();
         return true;
      }
      return false;
   }

   private:
      std::map<sig_atomic_t, void*>      m_signals;
      mutable std::mutex                 m_mutex;
 };


   template <sig_atomic_t signal> inline void SetlHandler (Logger  logger, TraceLevel
   traceLevel,  PostAction postAction)
  {
     std::string extaInfo = SignalType<signal>::ToString();
     extaInfo += " signal. \n";
     HandlerInfo info(logger, traceLevel, postAction, extaInfo);
     Handlers::Instance().Install<signal>(info);
  }

  template <sig_atomic_t signal> inline bool Handle()
  {
     bool status = Handlers::Instance().Handle<signal>();
     return status;
  }

}


inline void SignalHandler_SIGFPE(int)
{
   bool handled =signal_impl::Handle<SIGFPE>();
}

inline void SignalHandler_SIGSEGV(int)
{
   bool handled =signal_impl::Handle<SIGSEGV>();
}

inline void SetSIGFPEHandler(Logger logger, TraceLevel traceLevel =
TraceLevel::BACK_TRACE, PostAction postAction = PostAction::STD_DEFAULT )
{
      signal_impl::SetlHandler<SIGFPE>(logger, traceLevel, postAction );
      signal(SIGFPE, &SignalHandler_SIGFPE );
}

inline void  SetSIGSEGVHandler(Logger logger, TraceLevel traceLevel =
TraceLevel::BACK_TRACE, PostAction postAction = PostAction::STD_DEFAULT )
{
    signal_impl::SetlHandler<SIGSEGV>(logger, traceLevel, postAction );
    signal(SIGSEGV, &SignalHandler_SIGSEGV );
}



}
```

[redacted]

```
#ifndef CPP_TOOLS_EXCEPTION_H_
#define CPP_TOOLS_EXCEPTION_H_

#include <exception>
#include <stdexcept>
#include <string>
#include <memory>
#include <thread>
#include <mutex>
#include <functional>
#include "CPPTools/BackTracer.h"
#include "CPPTools/Signal.h"


namespace cpp_tools
{

 namespace exception_impl
 {

   struct TerminateHandler
{

      static TerminateHandler& Instance()
      {
         HandlerInfo  info(DummyLogger, TraceLevel::NO_TRACE ,  PostAction::NO_ACTION, "");
         static TerminateHandler handler(info);
         return handler;
      }
     void Handle()
     {
       std::lock_guard<std::mutex> l(m_mutex);
       ExceptionHandler(m_info);
     }

     void  SetHandlerInfo(HandlerInfo& info)
     {
        std::lock_guard<std::mutex> l(m_mutex);
        m_info = info;
     }
      private:
         TerminateHandler (HandlerInfo& info):m_info(info)
         {}
      private:
         HandlerInfo                      m_info;
         mutable std::mutex                m_mutex;
};

}

 inline void TerminateHandler()
 {
    exception_impl::TerminateHandler::Instance().Handle();
 }

 inline void SetTerminateHandler(Logger logger)
 {
    HandlerInfo info(logger, TraceLevel::EXCEPTION_TRACE,  PostAction::STD_DEFAULT,
    "Terminate handler");
    exception_impl::TerminateHandler::Instance().SetHandlerInfo(info);
    std::set_terminate(TerminateHandler);
 }

}
#endif  /* EXCEPTION_H_ */
```