java线程崩溃进程为什么不会被结束

典型回答

当进程在执行非法内存访问时，Linux内核会发出SIGSEGV信号，若进程没有注册信号处理函数会直接退出，并产生`Segment Fault`错误提示。
JVM在启动时注册了SIGSEGV信号处理函数，处理函数内部做了一些逻辑，如：如果是因为栈空间不够则抛出StackOverFlowError。

扩展知识

Linux为何会对访问内存错误发出异常信号?

Linux通过MMU（内存管理单元）将物理内存转换为虚拟内存，确保每个进程能看到的地址空间都是相同的，并且是自己进程独享的。32位下每个进程可访问从0到4G内存（用户空间3G，内核1G）。但实际上物理内存是所有进程共享的，所以内核需要有能力防止进程随意破坏不属于自己的内存空间。
使用C语言执行如下代码：

```
int main(int argc,char **argv){
    int *p =(int *)0xC0000FEE;
    *p = 1024;
    return 0;
}
```

以上代码由于非法向进程的内核地址空间`0xC0000FEE`处写入数据1024，操作系统会为该进程发出SIGSEGV的信号，由于我们没有实现信号处理函数，所以进程会默认被系统杀死，并向控制台输出`Segment Fault`错误。

进程、主线程、子线程的关系

在Linux下通过fork函数创建进程，fork创建进程时会将调用者（父进程）的页表、文件系统、打开文件句柄等信息复制一份，这是一个成本较大的操作，所以引入线程。Linux的线程也是通过fork函数实现，其入参与进程略有不同（CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND），这样使得所有子线程与进程共享页表、文件系统、打开句柄等信息，降低fork系统调用的成本。
进程和线程在Linux内核中都被抽象为一个task_struct的结构体，其中包括了页表、栈信息、文件描述符等很多信息，其中tgid保存了线程组id，如果tgid为pid则是所谓的主线程，而所有子线程tgid就是子线程的pid。本质上主线程和子线程没有任何区别（都是进程fork出来的，共享的信息

都一样）。
我们看到线程的fork参数中有一个CLONE_SIGHAND，代表线程与进程共享信号处理函数。所以**JVM内部在启动时注册的信号处理函数，会被所有线程复用。JVM注册的新号处理函数会抛出各种Error和Exception，但不会结束JVM进程。这样子线程崩溃了，也不会影响JVM中的其他线程。**

JVM都处理了哪些异常信号?

除了上面提到的SIGSEGV信号，JVM还注册了如下信号处理函数：

信号值	信号名称	作用	
11	SIGSEGV	试图访问未分配给自己的内存，或试图往没有写权限的内存地址写数据	
13	SIGPIPE	管道破裂。这个信号通常在进程间通信产生，比如采用 FIFO (管道) 通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到 SIGPIPE 信号。此外用 Socket 通信的两个进程，写进程在写 Socket 的时候，读进程已经终止。	
7	SIGBUS	非法地址，包括内存地址对齐 (alignment) 出错。比如访问一个四个字长的整数，但其地址不是 4 的倍数。它与 SIGSEGV 的区别在于后者是由于对合法存储地址的非法访问触发的 (如访问不属于自己存储空间或只读存储空间)。	
4	SIGILL	执行了非法指令。通常是因为可执行文件本身出现错误，或者试图执行数据段. 堆栈溢出时也有可能产生这个信号。	
8	SIGFPE	在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等其它所有的算术的错误。	
25	SIGXFSZ	当进程企图扩大文件以至于超过文件大小资源限制。	

JVM拦截这些信号的目的有三个：

1. 将异常转换成Error、Exception等信息抛给用户，如：StackOverFlowError、OutOfMemoryError、NullPointException等。
2. 在对系统产生致命错误之前做一些保护性动作。
3. 防止操作系统杀死JVM进程。

对于其他未注册信号处理函数的异常信号，JVM会退出，并生成 `hs_err_pid_xxx.log` crash文件用于分析异常。

从源码角度看看整体流程

最后我们从JVM的入口点向下查找注册信号处理函数的代码，由于篇幅原因省略无关代码，以下为jdk8-b08的源码，不同jdk版本信号处理逻辑略有不同。

```
// jdk/jdk/src/share/bin/main.c
int
main(int argc, char ** argv)
{
    int margc;
    char** margv;
    const jboolean const_javaw = JNI_FALSE;

    margc = argc;
    margv = argv;
#endif /* JAVAW */

// JLT_Launch函数开始初始化JVM虚拟机
    return JLI_Launch(margc, margv,
                   sizeof(const_jargs) / sizeof(char *),
                   ...);
}
```

```
// jdk/jdk/src/share/bin/java.c
int
JLI_Launch(int argc, char ** argv,              /* main argc, argc */
        int jargc, const char** jargv,          /* java args */
      ...
)
{
 //省略无关代码..
    if (!LoadJavaVM(jvmpath, &ifn)) { //加载JVM虚拟机
        return(6);
    }
 // ...
}
```

```
// jdk/hotspot/src/os/posix/launcher/java_md.c
jboolean
LoadJavaVM(const char *jvmpath, InvocationFunctions *ifn)
{
    //省略无关代码..
    //通过动态加载lib库的形式创建JVM
    ifn->CreateJavaVM = (CreateJavaVM_t)
```

```
      dlsym(libjvm, "JNI_CreateJavaVM");
    // ...
}
```

```
// jdk/hotspot/src/share/vm/prims/jni.cpp
_JNI_IMPORT_OR_EXPORT_ jint JNICALL JNI_CreateJavaVM(JavaVM **vm, void
**penv, void *args) {
  HS_DTRACE_PROBE3(hotspot_jni, CreateJavaVM__entry, vm, penv, args);
    //省略无关代码..
      result = Threads::create_vm((JavaVMInitArgs*) args, &can_try_again);
//创建JVM线程
    //...
 }
```

```
 // jdk/hotspot/src/share/vm/runtime/thread.cpp
 jint Threads::create_vm(JavaVMInitArgs* args, bool* canTryAgain) {
  //省略无关代码..
  // Initialize the os module after parsing the args
  jint os_init_2_result = os::init_2(); //调用与操作系统有关的init_2方法，不同操作
系统实现略有不同
  //...
}
```

```
// jdk/hotspot/src/os/linux/vm/os_linux.cpp
jint os::init_2(void)
{
    //省略无关代码..
    Linux::signal_sets_init(); //初始化信号集
    Linux::install_signal_handlers(); //注册信号处理函数，这里是重点
    //...
}

void os::Linux::install_signal_handlers() {
  if (!signal_handlers_are_installed) {
    signal_handlers_are_installed = true;
    //省略无关代码..
    //以下是注册上面表格中列出的信号
    set_signal_handler(SIGSEGV, true);
```

```
    set_signal_handler(SIGPIPE, true);
    set_signal_handler(SIGBUS, true);
    set_signal_handler(SIGILL, true);
    set_signal_handler(SIGFPE, true);
    set_signal_handler(SIGXFSZ, true);
   //...
   }
 }

void os::Linux::set_signal_handler(int sig, bool set_installed) {
 //省略无关代码..
  struct sigaction sigAct;
  sigfillset(&(sigAct.sa_mask));
  sigAct.sa_handler = SIG_DFL;
  if (!set_installed) {
    sigAct.sa_flags = SA_SIGINFO|SA_RESTART;
  } else {
    sigAct.sa_sigaction = signalHandler; //注册信号处理函数，这里是重点
    sigAct.sa_flags = SA_SIGINFO|SA_RESTART;
  }
    //...
}

extern "C" JNIEXPORT int
JVM_handle_linux_signal(int signo, siginfo_t* siginfo,
                        void* ucontext, int abort_if_unrecognized); //声明信
号处理逻辑，由外部文件导入

void signalHandler(int sig, siginfo_t* info, void* uc) {
  assert(info != NULL && uc != NULL, "it must be old kernel");
  JVM_handle_linux_signal(sig, info, uc, true); //执行信号处理逻辑
}
```

```
// jdk/hotspot/src/os_cpu/linux_x86/vm/os_linux_x86.cpp
extern "C" JNIEXPORT int
JVM_handle_linux_signal(int sig,
                        siginfo_t* info,
                        void* ucVoid,
                        int abort_if_unrecognized) {
    //这个函数包含了最终被信号处理函数拦截执行的逻辑，比较长，这里摘出栈溢出、打印crash文
```

件的代码

```
    //以下是StackOverFlowError的异常处理流程，这里可以看到JVM会根据当前栈空间范围和线程类型thread_state()为用户代码创建的线程，抛出StackOverFlowError
    if (sig == SIGSEGV) {
      address addr = (address) info->si_addr;

      // check if fault address is within thread stack
      if (addr < thread->stack_base() &&
          addr >= thread->stack_base() - thread->stack_size()) {
        // stack overflow
        if (thread->in_stack_yellow_zone(addr)) {
          thread->disable_stack_yellow_zone();
          if (thread->thread_state() == _thread_in_Java) {
            // Throw a stack overflow exception.  Guard pages will be reenabled
            // while unwinding the stack.
            stub = SharedRuntime::continuation_for_implicit_exception(thread, pc, SharedRuntime::STACK_OVERFLOW);
          } else {
            // Thread was in the vm or native code.  Return and try to finish.
            return 1;
          }
        }  //省略其他分支
    }
    //...
    //上面省略很多代码，在函数最后会打crash文件hs_err_pid_xxx.log
    VMError err(t, sig, pc, info, ucVoid);
    err.report_and_die();
  }
```