# Debugging on Linux for s/390 & z/Architecture

by
Denis Joseph Barrow (djbarrow@de.ibm.com,barrow_dj@yahoo.com)


Best viewed with fixed width fonts

## Overview of Document:

This document is intended to give a good overview of how to debug Linux for s/390 & z/Architecture. It isn't intended as a complete reference & not a tutorial on the fundamentals of C & assembly. It doesn't go into 390 IO in any detail. It is intended to complement the documents in the reference section below & any other worthwhile references you get.

It is intended like the Enterprise Systems Architecture/390 Reference Summary to be printed out & used as a quick cheat sheet self help style reference when problems occur.

## Contents



## Register Set

The current architectures have the following registers.

16  General propose registers, 32 bit on s/390 64 bit on z/Architecture, r0-r15 or gpr0-gpr15 used for arithmetic & addressing.

```
16 Control registers, 32 bit on s/390 64 bit on z/Architecture, ( cr0-cr15
kernel usage only ) used for memory management,
interrupt control,debugging control etc.

16 Access registers ( ar0-ar15 ) 32 bit on s/390 & z/Architecture
not used by normal programs but potentially could
be used as temporary storage. Their main purpose is their 1 to 1
association with general purpose registers and are used in
the kernel for copying data between kernel & user address spaces.
Access register 0 ( & access register 1 on z/Architecture ( needs 64 bit
pointer ) ) is currently used by the pthread library as a pointer to
the current running threads private area.

16 64 bit floating point registers (fp0-fp15 ) IEEE & HFP floating
point format compliant on G5 upwards & a Floating point control reg (FPC)
4  64 bit registers (fp0,fp2,fp4 & fp6) HFP only on older machines.
Note:
Linux (currently) always uses IEEE & emulates G5 IEEE format on older machines,
( provided the kernel is configured for this ).


The PSW is the most important register on the machine it
is 64 bit on s/390 & 128 bit on z/Architecture & serves the roles of
a program counter (pc), condition code register,memory space designator.
In IBM standard notation I am counting bit 0 as the MSB.
It has several advantages over a normal program counter
in that you can change address translation & program counter
in a single instruction. To change address translation,
e.g. switching address translation off requires that you
have a logical=physical mapping for the address you are
currently running at.

      Bit           Value
s/390 z/Architecture
0       0     Reserved ( must be 0 ) otherwise specification exception occurs.

1       1     Program Event Recording 1 PER enabled,
              PER is used to facilitate debugging e.g. single stepping.

2-4    2-4    Reserved ( must be 0 ).

5       5     Dynamic address translation 1=DAT on.

6       6     Input/Output interrupt Mask

7       7     External interrupt Mask used primarily for interprocessor
signalling &
              clock interrupts.

8-11  8-11    PSW Key used for complex memory protection mechanism not used
under linux

12      12    1 on s/390 0 on z/Architecture

13      13    Machine Check Mask 1=enable machine check interrupts
```

```
14      14    Wait State set this to 1 to stop the processor except for
interrupts & give
              time to other LPARS used in CPU idle in the kernel to increase
overall
              usage of processor resources.

15      15    Problem state ( if set to 1 certain instructions are disabled )
              all linux user programs run with this bit 1
              ( useful info for debugging under VM ).

16-17 16-17   Address Space Control

              00 Primary Space Mode when DAT on
              The linux kernel currently runs in this mode, CR1 is affiliated
with
              this mode & points to the primary segment table origin etc.

              01 Access register mode this mode is used in functions to
              copy data between kernel & user space.

              10 Secondary space mode not used in linux however CR7 the
              register affiliated with this mode is & this & normally
              CR13=CR7 to allow us to copy data between kernel & user space.
              We do this as follows:
              We set ar2 to 0 to designate its
              affiliated gpr ( gpr2 )to point to primary=kernel space.
              We set ar4 to 1 to designate its
              affiliated gpr ( gpr4 ) to point to secondary=home=user space
              & then essentially do a memcopy(gpr2,gpr4,size) to
              copy data between the address spaces, the reason we use home space
for the
              kernel & don't keep secondary space free is that code will not run
in
              secondary space.

              11 Home Space Mode all user programs run in this mode.
              it is affiliated with CR13.

18-19 18-19   Condition codes (CC)

20    20      Fixed point overflow mask if 1=FPU exceptions for this event
              occur ( normally 0 )

21    21      Decimal overflow mask if 1=FPU exceptions for this event occur
              ( normally 0 )

22    22      Exponent underflow mask if 1=FPU exceptions for this event occur
              ( normally 0 )

23    23      Significance Mask if 1=FPU exceptions for this event occur
              ( normally 0 )

24-31 24-30   Reserved Must be 0.

      31      Extended Addressing Mode
```

```
      32      Basic Addressing Mode
              Used to set addressing mode
              PSW 31   PSW 32
                0        0        24 bit
                0        1        31 bit
                1        1        64 bit

32             1=31 bit addressing mode 0=24 bit addressing mode (for backward
               compatibility), linux always runs with this bit set to 1

33-64          Instruction address.
      33-63    Reserved must be 0
      64-127   Address
               In 24 bits mode bits 64-103=0 bits 104-127 Address
               In 31 bits mode bits 64-96=0 bits 97-127 Address
               Note: unlike 31 bit mode on s/390 bit 96 must be zero
               when loading the address with LPSWE otherwise a
               specification exception occurs, LPSW is fully backward
               compatible.
```

Prefix Page(s)
--------------
This per cpu memory area is too intimately tied to the processor not to mention.
It exists between the real addresses 0-4096 on s/390 & 0-8192 z/Architecture & is exchanged
with a 1 page on s/390 or 2 pages on z/Architecture in absolute storage by the set
prefix instruction in linux'es startup.
This page is mapped to a different prefix for each processor in an SMP configuration
( assuming the os designer is sane of course :-) ).
Bytes 0-512 ( 200 hex ) on s/390 & 0-512,4096-4544,4604-5119 currently on z/Architecture
are used by the processor itself for holding such information as exception indications &
entry points for exceptions.
Bytes after 0xc00 hex are used by linux for per processor globals on s/390 & z/Architecture
( there is a gap on z/Architecture too currently between 0xc00 & 1000 which linux uses ).
The closest thing to this on traditional architectures is the interrupt
vector table. This is a good thing & does simplify some of the kernel coding
however it means that we now cannot catch stray NULL pointers in the
kernel without hard coded checks.

Address Spaces on Intel Linux
=============================

The traditional Intel Linux is approximately mapped as follows forgive
the ascii art.

```
0xFFFFFFFF 4GB Himem                        *****************
                                            *               *
                                            * Kernel Space  *
```

```
                                                *              *
                                                ****************
****************
User Space Himem (typically 0xC0000000 3GB )*  User Stack   *          *
     *
                                                ****************          *
     *
                                                *  Shared Libs  *          * Next
Process *
                                                ****************          *     to
     *
                                                *              *   <==   *     Run
     *  <==
                                                *  User Program *          *
     *
                                                *   Data BSS    *          *
     *
                                                *    Text       *          *
     *
                                                *   Sections    *          *
     *
0x00000000                                      ****************
****************
```

Now it is easy to see that on Intel it is quite easy to recognise a kernel address
as being one greater than user space himem ( in this case 0xC0000000).
& addresses of less than this are the ones in the current running program on this
processor ( if an smp box ).
If using the virtual machine ( VM ) as a debugger it is quite difficult to
know which user process is running as the address space you are looking at
could be from any process in the run queue.

The limitation of Intels addressing technique is that the linux
kernel uses a very simple real address to virtual addressing technique
of Real Address=Virtual Address-User Space Himem.
This means that on Intel the kernel linux can typically only address
Himem=0xFFFFFFFF-0xC0000000=1GB & this is all the RAM these machines
can typically use.
They can lower User Himem to 2GB or lower & thus be
able to use 2GB of RAM however this shrinks the maximum size
of User Space from 3GB to 2GB they have a no win limit of 4GB unless
they go to 64 Bit.

On 390 our limitations & strengths make us slightly different.
For backward compatibility we are only allowed use 31 bits (2GB)
of our 32 bit addresses, however, we use entirely separate address
spaces for the user & kernel.

This means we can support 2GB of non Extended RAM on s/390, & more
with the Extended memory management swap device &
currently 4TB of physical memory currently on z/Architecture.

Address Spaces on Linux for s/390 & z/Architecture
==================================================

Our addressing scheme is as follows

```
Himem 0x7fffffff 2GB on s/390    *****************          ****************
currently 0x3ffffffffff (2^42)-1 *  User Stack   *          *              *
on z/Architecture.               *****************          *              *
                                 *  Shared Libs  *          *              *

                                 *****************          *              *
                                 *               *          *    Kernel    *
                                 *  User Program *          *              *
                                 *   Data BSS    *          *              *
                                 *    Text       *          *              *
                                 *   Sections    *          *              *
0x00000000                       *****************          ****************
```

This also means that we need to look at the PSW problem state bit
or the addressing mode to decide whether we are looking at
user or kernel space.

Virtual Addresses on s/390 & z/Architecture
===========================================

A virtual address on s/390 is made up of 3 parts
The SX ( segment index, roughly corresponding to the PGD & PMD in linux terminology )
being bits 1-11.
The PX ( page index, corresponding to the page table entry (pte) in linux terminology )
being bits 12-19.
The remaining bits BX (the byte index are the offset in the page )
i.e. bits 20 to 31.

On z/Architecture in linux we currently make up an address from 4 parts.
The region index bits (RX) 0-32 we currently use bits 22-32
The segment index (SX) being bits 33-43
The page index (PX) being bits  44-51
The byte index (BX) being bits  52-63

Notes:
1) s/390 has no PMD so the PMD is really the PGD also.
A lot of this stuff is defined in pgtable.h.

2) Also seeing as s/390's page indexes are only 1k  in size
(bits 12-19 x 4 bytes per pte ) we use 1 ( page 4k )
to make the best use of memory by updating 4 segment indices
entries each time we mess with a PMD & use offsets
0,1024,2048 & 3072 in this page as for our segment indexes.
On z/Architecture our page indexes are now 2k in size
( bits 12-19 x 8 bytes per pte ) we do a similar trick
but only mess with 2 segment indices each time we mess with
a PMD.

3) As z/Architecture supports up to a massive 5-level page table lookup we
can only use 3 currently on Linux ( as this is all the generic kernel
currently supports ) however this may change in future
this allows us to access ( according to my sums )
4TB of virtual storage per process i.e.
4096*512(PTES)*1024(PMDS)*2048(PGD) = 4398046511104 bytes,
enough for another 2 or 3 of years I think :-).
to do this we use a region-third-table designation type in
our address space control registers.

The Linux for s/390 & z/Architecture Kernel Task Structure
==========================================================
Each process/thread under Linux for S390 has its own kernel task_struct
defined in linux/include/linux/sched.h
The S390 on initialisation & resuming of a process on a cpu sets
the __LC_KERNEL_STACK variable in the spare prefix area for this cpu
(which we use for per-processor globals).

The kernel stack pointer is intimately tied with the task structure for
each processor as follows.

```
                      s/390
            ************************
            *  1 page kernel stack *
            *        ( 4K )        *
            ************************
            *   1 page task_struct *
            *        ( 4K )        *
8K aligned  ************************

                 z/Architecture
            ************************
            *  2 page kernel stack *
            *        ( 8K )        *
            ************************
            *  2 page task_struct  *
            *        ( 8K )        *
16K aligned ************************
```

What this means is that we don't need to dedicate any register or global
variable
to point to the current running process & can retrieve it with the following
very simple construct for s/390 & one very similar for z/Architecture.

```
static inline struct task_struct * get_current(void)
{
        struct task_struct *current;
        __asm__("lhi   %0,-8192\n\t"
                "nr    %0,15"
                : "=r" (current) );
        return current;
}
```

i.e. just anding the current kernel stack pointer with the mask -8192.
Thankfully because Linux doesn't have support for nested IO interrupts

& our devices have large buffers can survive interrupts being shut for
short amounts of time we don't need a separate stack for interrupts.

Register Usage & Stackframes on Linux for s/390 & z/Architecture
=================================================================
Overview:
---------
This is the code that gcc produces at the top & the bottom of
each function. It usually is fairly consistent & similar from
function to function & if you know its layout you can probably
make some headway in finding the ultimate cause of a problem
after a crash without a source level debugger.

Note: To follow stackframes requires a knowledge of C or Pascal &
limited knowledge of one assembly language.

It should be noted that there are some differences between the
s/390 & z/Architecture stack layouts as the z/Architecture stack layout didn't
have
to maintain compatibility with older linkage formats.

Glossary:
---------
alloca:
This is a built in compiler function for runtime allocation
of extra space on the callers stack which is obviously freed
up on function exit ( e.g. the caller may choose to allocate nothing
of a buffer of 4k if required for temporary purposes ), it generates
very efficient code ( a few cycles ) when compared to alternatives
like malloc.

automatics: These are local variables on the stack,
i.e they aren't in registers & they aren't static.

back-chain:
This is a pointer to the stack pointer before entering a
framed functions ( see frameless function ) prologue got by
dereferencing the address of the current stack pointer,
i.e. got by accessing the 32 bit value at the stack pointers
current location.

base-pointer:
This is a pointer to the back of the literal pool which
is an area just behind each procedure used to store constants
in each function.

call-clobbered: The caller probably needs to save these registers if there
is something of value in them, on the stack or elsewhere before making a
call to another procedure so that it can restore it later.

epilogue:
The code generated by the compiler to return to the caller.

frameless-function
A frameless function in Linux for s390 & z/Architecture is one which doesn't
need more than the register save area ( 96 bytes on s/390, 160 on z/Architecture
)
given to it by the caller.
A frameless function never:
1) Sets up a back chain.
2) Calls alloca.
3) Calls other normal functions
4) Has automatics.

GOT-pointer:
This is a pointer to the global-offset-table in ELF
( Executable Linkable Format, Linux'es most common executable format ),
all globals & shared library objects are found using this pointer.

lazy-binding
ELF shared libraries are typically only loaded when routines in the shared
library are actually first called at runtime. This is lazy binding.

procedure-linkage-table
This is a table found from the GOT which contains pointers to routines
in other shared libraries which can't be called to by easier means.

prologue:
The code generated by the compiler to set up the stack frame.

outgoing-args:
This is extra area allocated on the stack of the calling function if the
parameters for the callee's cannot all be put in registers, the same
area can be reused by each function the caller calls.

routine-descriptor:
A COFF  executable format based concept of a procedure reference
actually being 8 bytes or more as opposed to a simple pointer to the routine.
This is typically defined as follows
Routine Descriptor offset 0=Pointer to Function
Routine Descriptor offset 4=Pointer to Table of Contents
The table of contents/TOC is roughly equivalent to a GOT pointer.
& it means that shared libraries etc. can be shared between several
environments each with their own TOC.

static-chain: This is used in nested functions a concept adopted from pascal
by gcc not used in ansi C or C++ ( although quite useful ), basically it
is a pointer used to reference local variables of enclosing functions.
You might come across this stuff once or twice in your lifetime.

e.g.
The function below should return 11 though gcc may get upset & toss warnings
about unused variables.

```
int FunctionA(int a)
{
	int b;
	FunctionC(int c)
	{
```

```
                b=c+1;
        }
        FunctionC(10);
        return(b);
}
```

s/390 & z/Architecture Register usage
=====================================

```
r0       used by syscalls/assembly                call-clobbered
r1       used by syscalls/assembly                call-clobbered
r2       argument 0 / return value 0              call-clobbered
r3       argument 1 / return value 1 (if long long) call-clobbered
r4       argument 2                               call-clobbered
r5       argument 3                               call-clobbered
r6       argument 4                               saved
r7       pointer-to arguments 5 to ...            saved
r8       this & that                              saved
r9       this & that                              saved
r10      static-chain ( if nested function )      saved
r11      frame-pointer ( if function used alloca ) saved
r12      got-pointer                              saved
r13      base-pointer                             saved
r14      return-address                           saved
r15      stack-pointer                            saved

f0       argument 0 / return value ( float/double ) call-clobbered
f2       argument 1                               call-clobbered
f4       z/Architecture argument 2                saved
f6       z/Architecture argument 3                saved
The remaining floating points
f1,f3,f5 f7-f15 are call-clobbered.
```

Notes:
------

1) The only requirement is that registers which are used
by the callee are saved, e.g. the compiler is perfectly
capable of using r11 for purposes other than a frame a
frame pointer if a frame pointer is not needed.
2) In functions with variable arguments e.g. printf the calling procedure
is identical to one without variable arguments & the same number of
parameters. However, the prologue of this function is somewhat more
hairy owing to it having to move these parameters to the stack to
get va_start, va_arg & va_end to work.
3) Access registers are currently unused by gcc but are used in
the kernel. Possibilities exist to use them at the moment for
temporary storage but it isn't recommended.
4) Only 4 of the floating point registers are used for
parameter passing as older machines such as G3 only have only 4
& it keeps the stack frame compatible with other compilers.
However with IEEE floating point emulation under linux on the
older machines you are free to use the other 12.
5) A long long or double parameter cannot be have the
first 4 bytes in a register & the second four bytes in the
outgoing args area. It must be purely in the outgoing args
area if crossing this boundary.

6) Floating point parameters are mixed with outgoing args
on the outgoing args area in the order the are passed in as parameters.
7) Floating point arguments 2 & 3 are saved in the outgoing args area for
z/Architecture

Stack Frame Layout
------------------
```
s/390     z/Architecture
0         0              back chain ( a 0 here signifies end of back chain )
4         8              eos ( end of stack, not used on Linux for S390 used in
other linkage formats )
8         16             glue used in other s/390 linkage formats for saved
routine descriptors etc.
12        24             glue used in other s/390 linkage formats for saved
routine descriptors etc.
16        32             scratch area
20        40             scratch area
24        48             saved r6 of caller function
28        56             saved r7 of caller function
32        64             saved r8 of caller function
36        72             saved r9 of caller function
40        80             saved r10 of caller function
44        88             saved r11 of caller function
48        96             saved r12 of caller function
52        104            saved r13 of caller function
56        112            saved r14 of caller function
60        120            saved r15 of caller function
64        128            saved f4 of caller function
72        132            saved f6 of caller function
80                       undefined
96        160            outgoing args passed from caller to callee
96+x      160+x          possible stack alignment ( 8 bytes desirable )
96+x+y    160+x+y        alloca space of caller ( if used )
96+x+y+z  160+x+y+z      automatics of caller ( if used )
0                        back-chain
```

A sample program with comments.
===============================

Comments on the function test
-----------------------------
1) It didn't need to set up a pointer to the constant pool gpr13 as it isn't
used
( :-( ).
2) This is a frameless function & no stack is bought.
3) The compiler was clever enough to recognise that it could return the
value in r2 as well as use it for the passed in parameter ( :-) ).
4) The basr ( branch relative & save ) trick works as follows the instruction
has a special case with r0,r0 with some instruction operands is understood as
the literal value 0, some risc architectures also do this ). So now
we are branching to the next address & the address new program counter is
in r13,so now we subtract the size of the function prologue we have executed
+ the size of the literal pool to get to the top of the literal pool
```
0040037c int test(int b)
{                                                              # Function prologue
```

```
below
  40037c:       90 de f0 34     stm     %r13,%r14,52(%r15) # Save registers r13
& r14
  400380:       0d d0           basr    %r13,%r0           # Set up pointer to
constant pool using
  400382:       a7 da ff fa     ahi     %r13,-6            # basr trick
        return(5+b);
                                                           # Huge main program
  400386:       a7 2a 00 05     ahi     %r2,5              # add 5 to r2

                                                           # Function epilogue
below
  40038a:       98 de f0 34     lm      %r13,%r14,52(%r15) # restore registers
r13 & 14
  40038e:       07 fe           br      %r14               # return
}
```

Comments on the function main
-----------------------------
1) The compiler did this function optimally ( 8-) )

Literal pool for main.

```
400390: ff ff ff ec     .long 0xffffffec
main(int argc,char *argv[])
{                                                          # Function prologue
below
  400394:       90 bf f0 2c     stm     %r11,%r15,44(%r15) # Save necessary
registers
  400398:       18 0f           lr      %r0,%r15           # copy stack pointer
to r0
  40039a:       a7 fa ff a0     ahi     %r15,-96           # Make area for
callee saving
  40039e:       0d d0           basr    %r13,%r0           # Set up r13 to point
to
  4003a0:       a7 da ff f0     ahi     %r13,-16           # literal pool
  4003a4:       50 00 f0 00     st      %r0,0(%r15)        # Save backchain

        return(test(5));                                   # Main Program Below
  4003a8:       58 e0 d0 00     l       %r14,0(%r13)       # load relative
address of test from
                                                           # literal pool
  4003ac:       a7 28 00 05     lhi     %r2,5              # Set first parameter
to 5
  4003b0:       4d ee d0 00     bas     %r14,0(%r14,%r13)  # jump to test
setting r14 as return
                                                           # address using
branch & save instruction.

                                                           # Function Epilogue
below
  4003b4:       98 bf f0 8c     lm      %r11,%r15,140(%r15)# Restore necessary
registers.
  4003b8:       07 fe           br      %r14               # return to do
program exit
}
```

Compiler updates
----------------

```
main(int argc,char *argv[])
{
  4004fc:       90 7f f0 1c             stm     %r7,%r15,28(%r15)
  400500:       a7 d5 00 04             bras    %r13,400508 <main+0xc>
  400504:       00 40 04 f4             .long   0x004004f4
  # compiler now puts constant pool in code to so it saves an instruction
  400508:       18 0f                   lr      %r0,%r15
  40050a:       a7 fa ff a0             ahi     %r15,-96
  40050e:       50 00 f0 00             st      %r0,0(%r15)
        return(test(5));
  400512:       58 10 d0 00             l       %r1,0(%r13)
  400516:       a7 28 00 05             lhi     %r2,5
  40051a:       0d e1                   basr    %r14,%r1
  # compiler adds 1 extra instruction to epilogue this is done to
  # avoid processor pipeline stalls owing to data dependencies on g5 &
  # above as register 14 in the old code was needed directly after being loaded
  # by the lm   %r11,%r15,140(%r15) for the br %14.
  40051c:       58 40 f0 98             l       %r4,152(%r15)
  400520:       98 7f f0 7c             lm      %r7,%r15,124(%r15)
  400524:       07 f4                   br      %r4
}
```

Hartmut ( our compiler developer ) also has been threatening to take out the stack backchain in optimised code as this also causes pipeline stalls, you have been warned.

64 bit z/Architecture code disassembly
--------------------------------------

If you understand the stuff above you'll understand the stuff below too so I'll avoid repeating myself & just say that some of the instructions have g's on the end of them to indicate they are 64 bit & the stack offsets are a bigger, the only other difference you'll find between 32 & 64 bit is that we now use f4 & f6 for floating point arguments on 64 bit.

```
00000000800005b0 <test>:
int test(int b)
{
        return(5+b);
    800005b0:   a7 2a 00 05             ahi     %r2,5
    800005b4:   b9 14 00 22             lgfr    %r2,%r2 # downcast to integer
    800005b8:   07 fe                   br      %r14
    800005ba:   07 07                   bcr     0,%r7


}

00000000800005bc <main>:
main(int argc,char *argv[])
{
    800005bc:   eb bf f0 58 00 24       stmg    %r11,%r15,88(%r15)
```

```
    800005c2:   b9 04 00 1f             lgr     %r1,%r15
    800005c6:   a7 fb ff 60             aghi    %r15,-160
    800005ca:   e3 10 f0 00 00 24       stg     %r1,0(%r15)
        return(test(5));
    800005d0:   a7 29 00 05             lghi    %r2,5
    # brasl allows jumps > 64k & is overkill here bras would do fune
    800005d4:   c0 e5 ff ff ff ee       brasl   %r14,800005b0 <test>
    800005da:   e3 40 f1 10 00 04       lg      %r4,272(%r15)
    800005e0:   eb bf f0 f8 00 04       lmg     %r11,%r15,248(%r15)
    800005e6:   07 f4                   br      %r4
}
```

## Compiling programs for debugging on Linux for s/390 & z/Architecture

-gdwarf-2 now works it should be considered the default debugging format for s/390 & z/Architecture as it is more reliable for debugging shared libraries,  normal -g debugging works much better now Thanks to the IBM java compiler developers bug reports.

This is typically done adding/appending the flags -g or -gdwarf-2 to the CFLAGS & LDFLAGS variables Makefile of the program concerned.

If using gdb & you would like accurate displays of registers & stack traces compile without optimisation i.e make sure that there is no -O2 or similar on the CFLAGS line of the Makefile & the emitted gcc commands, obviously this will produce worse code ( not advisable for shipment ) but it is an  aid to the debugging process.

This aids debugging because the compiler will copy parameters passed in in registers onto the stack so backtracing & looking at passed in parameters will work, however some larger programs which use inline functions will not compile without optimisation.

Debugging with optimisation has since much improved after fixing some bugs, please make sure you are using gdb-5.0 or later developed after Nov'2000.

## Figuring out gcc compile errors

If you are getting a lot of syntax errors compiling a program & the problem isn't blatantly obvious from the source.
It often helps to just preprocess the file, this is done with the -E option in gcc.
What this does is that it runs through the very first phase of compilation ( compilation in gcc is done in several stages & gcc calls many programs to achieve its end result ) with the -E option gcc just calls the gcc preprocessor (cpp).
The c preprocessor does the following, it joins all the files #included together recursively ( #include files can #include other files ) & also the c file you wish to compile.
It puts a fully qualified path of the #included files in a comment & it does macro expansion.
This is useful for debugging because
1) You can double check whether the files you expect to be included are the ones

that are being included ( e.g. double check that you aren't going to the i386 asm directory ).
2) Check that macro definitions aren't clashing with typedefs,
3) Check that definitions aren't being used before they are being included.
4) Helps put the line emitting the error under the microscope if it contains macros.

For convenience the Linux kernel's makefile will do preprocessing automatically for you
by suffixing the file you want built with .i ( instead of .o )

e.g.
from the linux directory type
make arch/s390/kernel/signal.i
this will build

```
s390-gcc -D__KERNEL__ -I/home1/barrow/linux/include -Wall -Wstrict-prototypes
-O2 -fomit-frame-pointer
-fno-strict-aliasing -D__SMP__ -pipe -fno-strength-reduce   -E
arch/s390/kernel/signal.c
> arch/s390/kernel/signal.i
```

Now look at signal.i you should see something like.

```
# 1 "/home1/barrow/linux/include/asm/types.h" 1
typedef unsigned short umode_t;
typedef __signed__ char __s8;
typedef unsigned char __u8;
typedef __signed__ short __s16;
typedef unsigned short __u16;
```

If instead you are getting errors further down e.g.
unknown instruction:2515 "move.l" or better still unknown instruction:2515
"Fixme not implemented yet, call Martin" you are probably are attempting to compile some code
meant for another architecture or code that is simply not implemented, with a fixme statement
stuck into the inline assembly code so that the author of the file now knows he has work to do.
To look at the assembly emitted by gcc just before it is about to call gas ( the gnu assembler )
use the -S option.
Again for your convenience the Linux kernel's Makefile will hold your hand &
do all this donkey work for you also by building the file with the .s suffix.
e.g.
from the Linux directory type
make arch/s390/kernel/signal.s

```
s390-gcc -D__KERNEL__ -I/home1/barrow/linux/include -Wall -Wstrict-prototypes
-O2 -fomit-frame-pointer
-fno-strict-aliasing -D__SMP__ -pipe -fno-strength-reduce  -S
arch/s390/kernel/signal.c
-o arch/s390/kernel/signal.s
```

This will output something like, ( please note the constant pool & the useful comments
in the prologue to give you a hand at interpreting it ).

```
.LC54:
        .string "misaligned (__u16 *) in __xchg\n"
.LC57:
        .string "misaligned (__u32 *) in __xchg\n"
.L$PG1: # Pool sys_sigsuspend
.LC192:
        .long   -262401
.LC193:
        .long   -1
.LC194:
        .long   schedule-.L$PG1
.LC195:
        .long   do_signal-.L$PG1
        .align 4
.globl sys_sigsuspend
        .type    sys_sigsuspend,@function
sys_sigsuspend:
#       leaf function           0
#       automatics              16
#       outgoing args           0
#       need frame pointer      0
#       call alloca             0
#       has varargs             0
#       incoming args (stack)   0
#       function length         168
        STM     8,15,32(15)
        LR      0,15
        AHI     15,-112
        BASR    13,0
.L$CO1: AHI     13,.L$PG1-.L$CO1
        ST      0,0(15)
        LR    8,2
        N     5,.LC192-.L$PG1(13)
```

Adding -g to the above output makes the output even more useful
e.g. typing
make CC:="s390-gcc -g" kernel/sched.s

which compiles.
s390-gcc -g -D__KERNEL__ -I/home/barrow/linux-2.3/include -Wall
-Wstrict-prototypes -O2 -fomit-frame-pointer -fno-strict-aliasing -pipe
-fno-strength-reduce   -S kernel/sched.c -o kernel/sched.s

also outputs stabs ( debugger ) info, from this info you can find out the
offsets & sizes of various elements in structures.
e.g. the stab for the structure

```
struct rlimit {
        unsigned long   rlim_cur;
        unsigned long   rlim_max;
};
```

is

```
.stabs "rlimit:T(151,2)=s8rlim_cur:(0,5),0,32;rlim_max:(0,5),32,32;;",128,0,0,0
```

from this stab you can see that
rlimit_cur starts at bit offset 0 & is 32 bits in size
rlimit_max starts at bit offset 32 & is 32 bits in size.

Debugging Tools:
================

objdump
=======
This is a tool with many options the most useful being ( if compiled with -g).
objdump --source <victim program or object file> > <victims debug listing >

The whole kernel can be compiled like this ( Doing this will make a 17MB kernel & a 200 MB listing ) however you have to strip it before building the image using the strip command to make it a more reasonable size to boot it.

A source/assembly mixed dump of the kernel can be done with the line
objdump --source vmlinux > vmlinux.lst
Also, if the file isn't compiled -g, this will output as much debugging information
as it can (e.g. function names). This is very slow as it spends lots
of time searching for debugging info. The following self explanatory line should be used
instead if the code isn't compiled -g, as it is much faster:
objdump --disassemble-all --syms vmlinux > vmlinux.lst

As hard drive space is valuable most of us use the following approach.
1) Look at the emitted psw on the console to find the crash address in the kernel.
2) Look at the file System.map ( in the linux directory ) produced when building

the kernel to find the closest address less than the current PSW to find the offending function.
3) use grep or similar to search the source tree looking for the source file
 with this function if you don't know where it is.
4) rebuild this object file with -g on, as an example suppose the file was
( /arch/s390/kernel/signal.o )
5) Assuming the file with the erroneous function is signal.c Move to the base of the
Linux source tree.
6) rm /arch/s390/kernel/signal.o
7) make /arch/s390/kernel/signal.o
8) watch the gcc command line emitted
9) type it in again or alternatively cut & paste it on the console adding the -g option.
10) objdump --source arch/s390/kernel/signal.o > signal.lst
This will output the source & the assembly intermixed, as the snippet below shows
This will unfortunately output addresses which aren't the same
as the kernel ones you should be able to get around the mental arithmetic
by playing with the --adjust-vma parameter to objdump.

```
static inline void spin_lock(spinlock_t *lp)
{
      a0:       18 34           lr      %r3,%r4
      a2:       a7 3a 03 bc     ahi     %r3,956
        __asm__ __volatile("    lhi   1,-1\n"
      a6:       a7 18 ff ff     lhi     %r1,-1
      aa:       1f 00           slr     %r0,%r0
      ac:       ba 01 30 00     cs      %r0,%r1,0(%r3)
      b0:       a7 44 ff fd     jm      aa <sys_sigsuspend+0x2e>
        saveset = current->blocked;
      b4:       d2 07 f0 68     mvc     104(8,%r15),972(%r4)
      b8:       43 cc
        return (set->sig[0] & mask) != 0;
}
```

6) If debugging under VM go down to that section in the document for more info.

I now have a tool which takes the pain out of --adjust-vma
& you are able to do something like
make /arch/s390/kernel/traps.lst
& it automatically generates the correctly relocated entries for
the text segment in traps.lst.
This tool is now standard in linux distro's in scripts/makelst

strace:
-------
Q. What is it ?
A. It is a tool for intercepting calls to the kernel & logging them
to a file & on the screen.

Q. What use is it ?
A. You can use it to find out what files a particular program opens.

Example 1
---------
If you wanted to know does ping work but didn't have the source
strace ping -c 1 127.0.0.1
& then look at the man pages for each of the syscalls below,
( In fact this is sometimes easier than looking at some spaghetti
source which conditionally compiles for several architectures ).
Not everything that it throws out needs to make sense immediately.

Just looking quickly you can see that it is making up a RAW socket
for the ICMP protocol.
Doing an alarm(10) for a 10 second timeout
& doing a gettimeofday call before & after each read to see
how long the replies took, & writing some text to stdout so the user
has an idea what is going on.

```
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) = 3
getuid()                                = 0
setuid(0)                               = 0
```

```
stat("/usr/share/locale/C/libc.cat", 0xbffff134) = -1 ENOENT (No such file or
directory)
stat("/usr/share/locale/libc/C", 0xbffff134) = -1 ENOENT (No such file or
directory)
stat("/usr/local/share/locale/C/libc.cat", 0xbffff134) = -1 ENOENT (No such file
or directory)
getpid()                                = 353
setsockopt(3, SOL_SOCKET, SO_BROADCAST, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_RCVBUF, [49152], 4) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(3, 1), ...}) = 0
mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40008000
ioctl(1, TCGETS, {B9600 opost isig icanon echo ...}) = 0
write(1, "PING 127.0.0.1 (127.0.0.1): 56 d"..., 42PING 127.0.0.1 (127.0.0.1): 56
data bytes
) = 42
sigaction(SIGINT, {0x8049ba0, [], SA_RESTART}, {SIG_DFL}) = 0
sigaction(SIGALRM, {0x8049600, [], SA_RESTART}, {SIG_DFL}) = 0
gettimeofday({948904719, 138951}, NULL) = 0
sendto(3, "\10\0D\201a\1\0\0\17#\2178\307\36"..., 64, 0, {sin_family=AF_INET,
sin_port=htons(0), sin_addr=inet_addr("127.0.0.1")}, 16) = 64
sigaction(SIGALRM, {0x8049600, [], SA_RESTART}, {0x8049600, [], SA_RESTART}) = 0
sigaction(SIGALRM, {0x8049ba0, [], SA_RESTART}, {0x8049600, [], SA_RESTART}) = 0
alarm(10)                               = 0
recvfrom(3, "E\0\0T\0005\0\0@\1|r\177\0\0\1\177"..., 192, 0,
{sin_family=AF_INET, sin_port=htons(50882), sin_addr=inet_addr("127.0.0.1")},
[16]) = 84
gettimeofday({948904719, 160224}, NULL) = 0
recvfrom(3, "E\0\0T\0006\0\0\377\1\275p\177\0"..., 192, 0,
{sin_family=AF_INET, sin_port=htons(50882), sin_addr=inet_addr("127.0.0.1")},
[16]) = 84
gettimeofday({948904719, 166952}, NULL) = 0
write(1, "64 bytes from 127.0.0.1: icmp_se"...,
5764 bytes from 127.0.0.1: icmp_seq=0 ttl=255 time=28.0 ms
```

Example 2
---------

```
strace passwd 2>&1 | grep open
produces the following output
open("/etc/ld.so.cache", O_RDONLY)      = 3
open("/opt/kde/lib/libc.so.5", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/lib/libc.so.5", O_RDONLY)        = 3
open("/dev", O_RDONLY)                  = 3
open("/var/run/utmp", O_RDONLY)         = 3
open("/etc/passwd", O_RDONLY)           = 3
open("/etc/shadow", O_RDONLY)           = 3
open("/etc/login.defs", O_RDONLY)       = 4
open("/dev/tty", O_RDONLY)              = 4
```

The 2>&1 is done to redirect stderr to stdout & grep is then filtering this
input
through the pipe for each line containing the string open.

Example 3
---------

```
Getting sophisticated
telnetd crashes & I don't know why

Steps
-----
1) Replace the following line in /etc/inetd.conf
telnet  stream  tcp     nowait  root    /usr/sbin/in.telnetd -h
with
telnet  stream  tcp     nowait  root    /blah

2) Create the file /blah with the following contents to start tracing telnetd
#!/bin/bash
/usr/bin/strace -o/t1 -f /usr/sbin/in.telnetd -h
3) chmod 700 /blah to make it executable only to root
4)
killall -HUP inetd
or ps aux | grep inetd
get inetd's process id
& kill -HUP inetd to restart it.

Important options
-----------------
-o is used to tell strace to output to a file in our case t1 in the root
directory
-f is to follow children i.e.
e.g in our case above telnetd will start the login process & subsequently a
shell like bash.
You will be able to tell which is which from the process ID's listed on the left
hand side
of the strace output.
-p<pid> will tell strace to attach to a running process, yup this can be done
provided
 it isn't being traced or debugged already & you have enough privileges,
the reason 2 processes cannot trace or debug the same program is that strace
becomes the parent process of the one being debugged & processes ( unlike people
)
can have only one parent.


However the file /t1 will get big quite quickly
to test it telnet 127.0.0.1

now look at what files in.telnetd execve'd
413   execve("/usr/sbin/in.telnetd", ["/usr/sbin/in.telnetd", "-h"], [/* 17 vars
*/]) = 0
414   execve("/bin/login", ["/bin/login", "-h", "localhost", "-p"], [/* 2 vars
*/]) = 0

Whey it worked!.


Other hints:
------------
If the program is not very interactive ( i.e. not much keyboard input )
& is crashing in one architecture but not in another you can do
an strace of both programs under as identical a scenario as you can
```

on both architectures outputting to a file then.
do a diff of the two traces using the diff program
i.e.
diff output1 output2
& maybe you'll be able to see where the call paths differed, this
is possibly near the cause of the crash.

## More info

Look at man pages for strace & the various syscalls
e.g. man strace, man alarm, man socket.

# Performance Debugging

gcc is capable of compiling in profiling code just add the -p option
to the CFLAGS, this obviously affects program size & performance.
This can be used by the gprof gnu profiling tool or the
gcov the gnu code coverage tool ( code coverage is a means of testing
code quality by checking if all the code in an executable in exercised by
a tester ).

## Using top to find out where processes are sleeping in the kernel

To do this copy the System.map from the root directory where
the linux kernel was built to the /boot directory on your
linux machine.
Start top
Now type fU<return>
You should see a new field called WCHAN which
tells you where each process is sleeping here is a typical output.

```
 6:59pm  up 41 min,  1 user,  load average: 0.00, 0.00, 0.00
28 processes: 27 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  0.0% user,  0.1% system,  0.0% nice, 99.8% idle
Mem:   254900K av,   45976K used,  208924K free,       0K shrd,   28636K buff
Swap:       0K av,       0K used,       0K free                    8620K cached

  PID USER     PRI  NI  SIZE  RSS SHARE WCHAN     STAT  LIB %CPU %MEM   TIME
COMMAND
  750 root      12   0   848  848   700 do_select S       0  0.1  0.3   0:00
in.telnetd
  767 root      16   0  1140 1140   964           R       0  0.1  0.4   0:00 top
    1 root       8   0   212  212   180 do_select S       0  0.0  0.0   0:00
init
    2 root       9   0     0    0     0 down_inte SW      0  0.0  0.0   0:00
kmcheck
```

## The time command

Another related command is the time command which gives you an indication
of where a process is spending the majority of its time.
e.g.
time ping -c 5 nc
outputs

```
real    0m4.054s
user    0m0.010s
sys     0m0.010s
```

Debugging under VM
==================

Notes
-----
Addresses & values in the VM debugger are always hex never decimal
Address ranges are of the format <HexValue1>-<HexValue2> or
<HexValue1>.<HexValue2>
e.g. The address range  0x2000 to 0x3000 can be described as 2000-3000 or
2000.1000

The VM Debugger is case insensitive.

VM's strengths are usually other debuggers weaknesses you can get at any
resource
no matter how sensitive e.g. memory management resources,change address
translation
in the PSW. For kernel hacking you will reap dividends if you get good at it.

The VM Debugger displays operators but not operands, probably because some
of it was written when memory was expensive & the programmer was probably proud
that
it fitted into 2k of memory & the programmers & didn't want to shock hardcore
VM'ers by
changing the interface :-), also the debugger displays useful information on the
same line &
the author of the code probably felt that it was a good idea not to go over
the 80 columns on the screen.

As some of you are probably in a panic now this isn't as unintuitive as it may
seem
as the 390 instructions are easy to decode mentally & you can make a good guess
at a lot
of them as all the operands are nibble ( half byte aligned ) & if you have an
objdump listing
also it is quite easy to follow, if you don't have an objdump listing keep a
copy of
the s/390 Reference Summary & look at between pages 2 & 7 or alternatively the
s/390 principles of operation.
e.g. even I can guess that

```
0001AFF8' LR    180F        CC 0
```

is a ( load register ) lr r0,r15

Also it is very easy to tell the length of a 390 instruction from the 2 most
significant
bits in the instruction ( not that this info is really useful except if you are
trying to
make sense of a hexdump of code ).
Here is a table

```
Bits                    Instruction Length
------------------------------------------
00                          2 Bytes
```

```
01                            4 Bytes
10                            4 Bytes
11                            6 Bytes
```

The debugger also displays other useful info on the same line such as the
addresses being operated on destination addresses of branches & condition codes.
e.g.

```
00019736' AHI   A7DAFF0E    CC 1
000198BA' BRC   A7840004 -> 000198C2'   CC 0
000198CE' STM   900EF068 >> 0FA95E78    CC 2
```

Useful VM debugger commands
---------------------------

I suppose I'd better mention this before I start
to list the current active traces do
Q TR
there can be a maximum of 255 of these per set
( more about trace sets later ).
To stop traces issue a
TR END.
To delete a particular breakpoint issue
TR DEL <breakpoint number>

The PA1 key drops to CP mode so you can issue debugger commands,
Doing alt c (on my 3270 console at least ) clears the screen.
hitting b <enter> comes back to the running operating system
from cp mode ( in our case linux ).
It is typically useful to add shortcuts to your profile.exec file
if you have one ( this is roughly equivalent to autoexec.bat in DOS ).
file here are a few from mine.

```
/* this gives me command history on issuing f12 */
set pf12 retrieve
/* this continues */
set pf8 imm b
/* goes to trace set a */
set pf1 imm tr goto a
/* goes to trace set b */
set pf2 imm tr goto b
/* goes to trace set c */
set pf3 imm tr goto c
```

Instruction Tracing
-------------------
Setting a simple breakpoint
TR I PSWA <address>
To debug a particular function try
TR I R <function address range>
TR I on its own will single step.

TR I DATA <MNEMONIC> <OPTIONAL RANGE> will trace for particular mnemonics
e.g.
TR I DATA 4D R 0197BC.4000
will trace for BAS'es ( opcode 4D ) in the range 0197BC.4000
if you were inclined you could add traces for all branch instructions &
suffix them with the run prefix so you would have a backtrace on screen
when a program crashes.
TR BR <INTO OR FROM> will trace branches into or out of an address.
e.g.
TR BR INTO 0 is often quite useful if a program is getting awkward & deciding
to branch to 0 & crashing as this will stop at the address before in jumps to 0.
TR I R <address range> RUN cmd d g
single steps a range of addresses but stays running &
displays the gprs on each step.

Displaying & modifying Registers
--------------------------------
D G will display all the gprs
Adding a extra G to all the commands is necessary to access the full 64 bit
content in VM on z/Architecture obviously this isn't required for access
registers
as these are still 32 bit.
e.g. DGG instead of DG
D X will display all the control registers
D AR will display all the access registers
D AR4-7 will display access registers 4 to 7
CPU ALL D G will display the GRPS of all CPUS in the configuration
D PSW will display the current PSW
st PSW 2000 will put the value 2000 into the PSW &
cause crash your machine.
D PREFIX displays the prefix offset

Displaying Memory
-----------------
To display memory mapped using the current PSW's mapping try
D <range>
To make VM display a message each time it hits a particular address & continue
try
D I<range> will disassemble/display a range of instructions.
ST addr 32 bit word will store a 32 bit aligned address
D T<range> will display the EBCDIC in an address ( if you are that way inclined
)
D R<range> will display real addresses ( without DAT ) but with prefixing.
There are other complex options to display if you need to get at say home space
but are in primary space the easiest thing to do is to temporarily
modify the PSW to the other addressing mode, display the stuff & then
restore it.

Hints
-----
If you want to issue a debugger command without halting your virtual machine

with the
PA1 key try prefixing the command with #CP e.g.
#cp tr i pswa 2000
also suffixing most debugger commands with RUN will cause them not
to stop just display the mnemonic at the current instruction on the console.
If you have several breakpoints you want to put into your program &
you get fed up of cross referencing with System.map
you can do the following trick for several symbols.
grep do_signal System.map
which emits the following among other things
0001f4e0 T do_signal
now you can do

TR I PSWA 0001f4e0 cmd msg * do_signal
This sends a message to your own console each time do_signal is entered.
( As an aside I wrote a perl script once which automatically generated a REXX
script with breakpoints on every kernel procedure, this isn't a good idea
because there are thousands of these routines & VM can only set 255 breakpoints
at a time so you nearly had to spend as long pruning the file down as you would
entering the msg's by hand ),however, the trick might be useful for a single
object file.
On linux'es 3270 emulator x3270 there is a very useful option under the file
ment
Save Screens In File this is very good of keeping a copy of traces.

From CMS help <command name> will give you online help on a particular command.
e.g.
HELP DISPLAY

Also CP has a file called profile.exec which automatically gets called
on startup of CMS ( like autoexec.bat ), keeping on a DOS analogy session
CP has a feature similar to doskey, it may be useful for you to
use profile.exec to define some keystrokes.
e.g.
SET PF9 IMM B
This does a single step in VM on pressing F8.
SET PF10 ^
This sets up the ^ key.
which can be used for ^c (ctrl-c),^z (ctrl-z) which can't be typed directly into
some 3270 consoles.
SET PF11 ^-
This types the starting keystrokes for a sysrq see SysRq below.
SET PF12 RETRIEVE
This retrieves command history on pressing F12.

Sometimes in VM the display is set up to scroll automatically this
can be very annoying if there are messages you wish to look at
to stop this do
TERM MORE 255 255
This will nearly stop automatic screen updates, however it will
cause a denial of service if lots of messages go to the 3270 console,
so it would be foolish to use this as the default on a production machine.

Tracing particular processes

-----------------------------
The kernel's text segment is intentionally at an address in memory that it will
very seldom collide with text segments of user programs ( thanks Martin ),
this simplifies debugging the kernel.
However it is quite common for user processes to have addresses which collide
this can make debugging a particular process under VM painful under normal
circumstances as the process may change when doing a
TR I R <address range>.
Thankfully after reading VM's online help I figured out how to debug
I particular process.

Your first problem is to find the STD ( segment table designation )
of the program you wish to debug.
There are several ways you can do this here are a few
1) objdump --syms <program to be debugged> | grep main
To get the address of main in the program.
tr i pswa <address of main>
Start the program, if VM drops to CP on what looks like the entry
point of the main function this is most likely the process you wish to debug.
Now do a D X13 or D XG13 on z/Architecture.
On 31 bit the STD is bits 1-19 ( the STO segment table origin )
& 25-31 ( the STL segment table length ) of CR13.
now type
TR I R STD <CR13's value> 0.7fffffff
e.g.
TR I R STD 8F32E1FF 0.7fffffff
Another very useful variation is
TR STORE INTO STD <CR13's value> <address range>
for finding out when a particular variable changes.

An alternative way of finding the STD of a currently running process
is to do the following, ( this method is more complex but
could be quite convenient if you aren't updating the kernel much &
so your kernel structures will stay constant for a reasonable period of
time ).

```
grep task /proc/<pid>/status
from this you should see something like
task: 0f160000 ksp: 0f161de8 pt_regs: 0f161f68
This now gives you a pointer to the task structure.
Now make CC:="s390-gcc -g" kernel/sched.s
To get the task_struct stabinfo.
( task_struct is defined in include/linux/sched.h ).
Now we want to look at
task->active_mm->pgd
on my machine the active_mm in the task structure stab is
active_mm:(4,12),672,32
its offset is 672/8=84=0x54
the pgd member in the mm_struct stab is
pgd:(4,6)=*(29,5),96,32
so its offset is 96/8=12=0xc

so we'll
hexdump -s 0xf160054 /dev/mem | more
i.e. task_struct+active_mm offset
to look at the active_mm member
```

f160054 0fee cc60 0019 e334 0000 0000 0000 0011
hexdump -s 0x0feecc6c /dev/mem | more
i.e. active_mm+pgd offset
feecc6c 0f2c 0000 0000 0001 0000 0001 0000 0010
we get something like
now do
TR I R STD <pgd|0x7f> 0.7fffffff
i.e. the 0x7f is added because the pgd only
gives the page table origin & we need to set the low bits
to the maximum possible segment table length.
TR I R STD 0f2c007f 0.7fffffff
on z/Architecture you'll probably need to do
TR I R STD <pgd|0x7> 0.ffffffffffffffff
to set the TableType to 0x1 & the Table length to 3.

## Tracing Program Exceptions

If you get a crash which says something like
illegal operation or specification exception followed by a register dump
You can restart linux & trace these using the tr prog <range or value> trace option.

The most common ones you will normally be tracing for is
1=operation exception
2=privileged operation exception
4=protection exception
5=addressing exception
6=specification exception
10=segment translation exception
11=page translation exception

The full list of these is on page 22 of the current s/390 Reference Summary.
e.g.
tr prog 10 will trace segment translation exceptions.
tr prog on its own will trace all program interruption codes.

## Trace Sets

On starting VM you are initially in the INITIAL trace set.
You can do a Q TR to verify this.
If you have a complex tracing situation where you wish to wait for instance
till a driver is open before you start tracing IO, but know in your
heart that you are going to have to make several runs through the code till you
have a clue whats going on.

What you can do is
TR I PSWA <Driver open address>
hit b to continue till breakpoint
reach the breakpoint
now do your
TR GOTO B
TR IO 7c08-7c09 inst int run

or whatever the IO channels you wish to trace are & hit b

To got back to the initial trace set do
TR GOTO INITIAL
& the TR I PSWA <Driver open address> will be the only active breakpoint again.

## Tracing linux syscalls under VM

Syscalls are implemented on Linux for S390 by the Supervisor call instruction (SVC) there 256
possibilities of these as the instruction is made up of a 0xA opcode & the second byte being
the syscall number. They are traced using the simple command.
TR SVC <Optional value or range>
the syscalls are defined in linux/arch/s390/include/asm/unistd.h
e.g. to trace all file opens just do
TR SVC 5 ( as this is the syscall number of open )

## SMP Specific commands

To find out how many cpus you have
Q CPUS displays all the CPU's available to your virtual machine
To find the cpu that the current cpu VM debugger commands are being directed at do
Q CPU to change the current cpu VM debugger commands are being directed at do
CPU <desired cpu no>

On a SMP guest issue a command to all CPUs try prefixing the command with cpu all.
To issue a command to a particular cpu try cpu <cpu number> e.g.
CPU 01 TR I R 2000.3000
If you are running on a guest with several cpus & you have a IO related problem
& cannot follow the flow of code but you know it isn't smp related.
from the bash prompt issue
shutdown -h now or halt.
do a Q CPUS to find out how many cpus you have
detach each one of them from cp except cpu 0
by issuing a
DETACH CPU 01-(number of cpus in configuration)
& boot linux again.
TR SIGP will trace inter processor signal processor instructions.
DEFINE CPU 01-(number in configuration)
will get your guests cpus back.

## Help for displaying ascii textstrings

On the very latest VM Nucleus'es VM can now display ascii
( thanks Neale for the hint ) by doing
D TX<lowaddr>.<len>
e.g.
D TX0.100

Alternatively

=============
Under older VM debuggers ( I love EBDIC too ) you can use this little program I wrote which
will convert a command line of hex digits to ascii text which can be compiled under linux &
you can copy the hex digits from your x3270 terminal to your xterm if you are debugging
from a linuxbox.

This is quite useful when looking at a parameter passed in as a text string
under VM ( unless you are good at decoding ASCII in your head ).

e.g. consider tracing an open syscall
TR SVC 5
We have stopped at a breakpoint
000151B0' SVC   0A05     -> 0001909A'   CC 0

D 20.8 to check the SVC old psw in the prefix area & see was it from userspace
( for the layout of the prefix area consult P18 of the s/390 390 Reference Summary
if you have it available ).
V00000020  070C2000 800151B2
The problem state bit wasn't set &  it's also too early in the boot sequence
for it to be a userspace SVC if it was we would have to temporarily switch the
psw to user space addressing so we could get at the first parameter of the open in
gpr2.
Next do a
D G2
GPR  2 =  00014CB4
Now display what gpr2 is pointing to
D 00014CB4.20
V00014CB4  2F646576 2F636F6E 736F6C65 00001BF5
V00014CC4  FC00014C B4001001 E0001000 B8070707
Now copy the text till the first 00 hex ( which is the end of the string
to an xterm & do hex2ascii on it.
hex2ascii 2F646576 2F636F6E 736F6C65 00
outputs
Decoded Hex:=/ d e v / c o n s o l e 0x00
We were opening the console device,

You can compile the code below yourself for practice :-),

```
/*
 *    hex2ascii.c
 *    a useful little tool for converting a hexadecimal command line to ascii
 *
 *    Author(s): Denis Joseph Barrow (djbarrow@de.ibm.com,barrow_dj@yahoo.com)
 *    (C) 2000 IBM Deutschland Entwicklung GmbH, IBM Corporation.
 */
#include <stdio.h>

int main(int argc,char *argv[])
{
  int cnt1,cnt2,len,toggle=0;
  int startcnt=1;
  unsigned char c,hex;
```

```
  if(argc>1&&(strcmp(argv[1],"-a")==0))
     startcnt=2;
  printf("Decoded Hex:=");
  for(cnt1=startcnt;cnt1<argc;cnt1++)
  {
    len=strlen(argv[cnt1]);
    for(cnt2=0;cnt2<len;cnt2++)
    {
       c=argv[cnt1][cnt2];
       if(c>='0'&&c<='9')
          c=c-'0';
       if(c>='A'&&c<='F')
          c=c-'A'+10;
       if(c>='a'&&c<='f')
          c=c-'a'+10;
       switch(toggle)
       {
          case 0:
             hex=c<<4;
             toggle=1;
          break;
          case 1:
             hex+=c;
             if(hex<32||hex>127)
             {
                if(startcnt==1)
                   printf("0x%02X ",(int)hex);
                else
                   printf(".");
             }
             else
             {
               printf("%c",hex);
               if(startcnt==1)
                  printf(" ");
             }
             toggle=0;
          break;
       }
    }
  }
  printf("\n");
}
```

Stack tracing under VM
----------------------
A basic backtrace
-----------------

Here are the tricks I use 9 out of 10 times it works pretty well,

When your backchain reaches a dead end

---------------------------------------
This can happen when an exception happens in the kernel & the kernel is entered twice
if you reach the NULL pointer at the end of the back chain you should be
able to sniff further back if you follow the following tricks.
1) A kernel address should be easy to recognise since it is in
primary space & the problem state bit isn't set & also
The Hi bit of the address is set.
2) Another backchain should also be easy to recognise since it is an
address pointing to another address approximately 100 bytes or 0x70 hex
behind the current stackpointer.

Here is some practice.
boot the kernel & hit PA1 at some random time
d g to display the gprs, this should display something like
```
GPR  0 =  00000001  00156018  0014359C  00000000
GPR  4 =  00000001  001B8888  000003E0  00000000
GPR  8 =  00100080  00100084  00000000  000FE000
GPR 12 =  00010400  8001B2DC  8001B36A  000FFED8
```
Note that GPR14 is a return address but as we are real men we are going to
trace the stack.
display 0x40 bytes after the stack pointer.

```
V000FFED8  000FFF38 8001B838 80014C8E 000FFF38
V000FFEE8  00000000 00000000 000003E0 00000000
V000FFEF8  00100080 00100084 00000000 000FE000
V000FFF08  00010400 8001B2DC 8001B36A 000FFED8
```

Ah now look at whats in sp+56 (sp+0x38) this is 8001B36A our saved r14 if
you look above at our stackframe & also agrees with GPR14.

now backchain
d 000FFF38.40
we now are taking the contents of SP to get our first backchain.

```
V000FFF38  000FFFA0 00000000 00014995 00147094
V000FFF48  00147090 001470A0 000003E0 00000000
V000FFF58  00100080 00100084 00000000 001BF1D0
V000FFF68  00010400 800149BA 80014CA6 000FFF38
```

This displays a 2nd return address of 80014CA6

now do d 000FFFA0.40 for our 3rd backchain

```
V000FFFA0  04B52002 0001107F 00000000 00000000
V000FFFB0  00000000 00000000 FF000000 0001107F
V000FFFC0  00000000 00000000 00000000 00000000
V000FFFD0  00010400 80010802 8001085A 000FFFA0
```

our 3rd return address is 8001085A

as the 04B52002 looks suspiciously like rubbish it is fair to assume that the
kernel entry routines

for the sake of optimisation don't set up a backchain.

now look at System.map to see if the addresses make any sense.

grep -i 0001b3 System.map
outputs among other things
0001b304 T cpu_idle
so 8001B36A
is cpu_idle+0x66 ( quiet the cpu is asleep, don't wake it )

grep -i 00014 System.map
produces among other things
00014a78 T start_kernel
so 0014CA6 is start_kernel+some hex number I can't add in my head.

grep -i 00108 System.map
this produces
00010800 T _stext
so   8001085A is _stext+0x5a

Congrats you've done your first backchain.

s/390 & z/Architecture IO Overview
==================================

I am not going to give a course in 390 IO architecture as this would take me quite a
while & I'm no expert. Instead I'll give a 390 IO architecture summary for Dummies if you have
the s/390 principles of operation available read this instead. If nothing else you may find a few
useful keywords in here & be able to use them on a web search engine like altavista to find
more useful information.

Unlike other bus architectures modern 390 systems do their IO using mostly
fibre optics & devices such as tapes & disks can be shared between several mainframes,
also S390 can support up to 65536 devices while a high end PC based system might be choking
with around 64. Here is some of the common IO terminology

Subchannel:
This is the logical number most IO commands use to talk to an IO device there can be up to
0x10000 (65536) of these in a configuration typically there is a few hundred. Under VM
for simplicity they are allocated contiguously, however on the native hardware they are not
they typically stay consistent between boots provided no new hardware is inserted or removed.
Under Linux for 390 we use these as IRQ's & also when issuing an IO command (CLEAR SUBCHANNEL,

HALT SUBCHANNEL,MODIFY SUBCHANNEL,RESUME SUBCHANNEL,START SUBCHANNEL,STORE SUBCHANNEL &
TEST SUBCHANNEL ) we use this as the ID of the device we wish to talk to, the most
important of these instructions are START SUBCHANNEL ( to start IO ), TEST SUBCHANNEL ( to check
whether the IO completed successfully ), & HALT SUBCHANNEL ( to kill IO ), a subchannel
can have up to 8 channel paths to a device this offers redundancy if one is not available.

Device Number:
This number remains static & Is closely tied to the hardware, there are 65536 of these
also they are made up of a CHPID ( Channel Path ID, the most significant 8 bits )
& another lsb 8 bits. These remain static even if more devices are inserted or removed
from the hardware, there is a 1 to 1 mapping between Subchannels & Device Numbers provided
devices aren't inserted or removed.

Channel Control Words:
CCWS are linked lists of instructions initially pointed to by an operation request block (ORB),
which is initially given to Start Subchannel (SSCH) command along with the subchannel number
for the IO subsystem to process while the CPU continues executing normal code.
These come in two flavours, Format 0 ( 24 bit for backward )
compatibility & Format 1 ( 31 bit ). These are typically used to issue read & write
( & many other instructions ) they consist of a length field & an absolute address field.
For each IO typically get 1 or 2 interrupts one for channel end ( primary status ) when the
channel is idle & the second for device end ( secondary status ) sometimes you get both
concurrently, you check how the IO went on by issuing a TEST SUBCHANNEL at each interrupt,
from which you receive an Interruption response block (IRB). If you get channel & device end
status in the IRB without channel checks etc. your IO probably went okay. If you didn't you
probably need a doctor to examine the IRB & extended status word etc.
If an error occurs, more sophisticated control units have a facility known as
concurrent sense this means that if an error occurs Extended sense information will
be presented in the Extended status word in the IRB if not you have to issue a
subsequent SENSE CCW command after the test subchannel.

TPI( Test pending interrupt) can also be used for polled IO but in multitasking multiprocessor
systems it isn't recommended except for checking special cases ( i.e. non looping checks for

pending IO etc. ).

Store Subchannel & Modify Subchannel can be used to examine & modify operating characteristics
of a subchannel ( e.g. channel paths ).

Other IO related Terms:
Sysplex: S390's Clustering Technology
QDIO: S390's new high speed IO architecture to support devices such as gigabit ethernet,
this architecture is also designed to be forward compatible with up & coming 64 bit machines.

General Concepts

Input Output Processors (IOP's) are responsible for communicating between
the mainframe CPU's & the channel & relieve the mainframe CPU's from the
burden of communicating with IO devices directly, this allows the CPU's to
concentrate on data processing.

IOP's can use one or more links ( known as channel paths ) to talk to each
IO device. It first checks for path availability & chooses an available one,
then starts ( & sometimes terminates IO ).
There are two types of channel path: ESCON & the Parallel IO interface.

IO devices are attached to control units, control units provide the
logic to interface the channel paths & channel path IO protocols to
the IO devices, they can be integrated with the devices or housed separately
& often talk to several similar devices ( typical examples would be raid
controllers or a control unit which connects to 1000 3270 terminals ).

```
    +---------------------------------------------------------------+
    | +-----+ +-----+ +-----+ +-----+  +----------+  +----------+   |
    | | CPU | | CPU | | CPU | | CPU |  |  Main    |  | Expanded |   |
    | |     | |     | |     | |     |  |  Memory  |  |  Storage |   |
    | +-----+ +-----+ +-----+ +-----+  +----------+  +----------+   |
    |---------------------------------------------------------------+
    |   IOP        |      IOP       |       IOP                     |
    |---------------------------------------------------------------
    | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C | C |
    ----------------------------------------------------------------
         ||                                              ||
         ||  Bus & Tag Channel Path                      || ESCON
         ||  ======================                      || Channel
         ||  ||                 ||                       || Path
    +----------+               +----------+         +----------+
    |          |               |          |         |          |
    |    CU    |               |    CU    |         |    CU    |
    |          |               |          |         |          |
    +----------+               +----------+         +----------+
       |      |                      |                 |       |
+----------+ +----------+      +----------+  +----------+ +----------+
|I/O Device| |I/O Device|      |I/O Device|  |I/O Device| |I/O Device|
+----------+ +----------+      +----------+  +----------+ +----------+
```

CPU = Central Processing Unit
C = Channel
IOP = IP Processor
CU = Control Unit

The 390 IO systems come in 2 flavours the current 390 machines support both

The Older 360 & 370 Interface,sometimes called the Parallel I/O interface, sometimes called Bus-and Tag & sometimes Original Equipment Manufacturers Interface (OEMI).

This byte wide Parallel channel path/bus has parity & data on the "Bus" cable & control lines on the "Tag" cable. These can operate in byte multiplex mode for sharing between several slow devices or burst mode & monopolize the channel for the whole burst. Up to 256 devices can be addressed on one of these cables. These cables are about one inch in diameter. The maximum unextended length supported by these cables is 125 Meters but this can be extended up to 2km with a fibre optic channel extended such as a 3044. The maximum burst speed supported is 4.5 megabytes per second however some really old processors support only transfer rates of 3.0, 2.0 & 1.0 MB/sec. One of these paths can be daisy chained to up to 8 control units.

ESCON if fibre optic it is also called FICON
Was introduced by IBM in 1990. Has 2 fibre optic cables & uses either leds or lasers for communication at a signaling rate of up to 200 megabits/sec. As 10bits are transferred for every 8 bits info this drops to 160 megabits/sec & to 18.6 Megabytes/sec once control info & CRC are added. ESCON only operates in burst mode.

ESCONs typical max cable length is 3km for the led version & 20km for the laser version known as XDF ( extended distance facility ). This can be further extended by using an ESCON director which triples the above mentioned ranges. Unlike Bus & Tag as ESCON is serial it uses a packet switching architecture the standard Bus & Tag control protocol is however present within the packets. Up to 256 devices can be attached to each control unit that uses one of these interfaces.

Common 390 Devices include:
Network adapters typically OSA2,3172's,2116's & OSA-E gigabit ethernet adapters,
Consoles 3270 & 3215 ( a teletype emulated under linux for a line mode console ).
DASD's direct access storage devices ( otherwise known as hard disks ).
Tape Drives.
CTC ( Channel to Channel Adapters ),
ESCON or Parallel Cables used as a very high speed serial link

between 2 machines. We use 2 cables under linux to do a bi-directional serial link.

Debugging IO on s/390 & z/Architecture under VM
===============================================

Now we are ready to go on with IO tracing commands under VM

A few self explanatory queries:
Q OSA
Q CTC
Q DISK ( This command is CMS specific )
Q DASD

Q OSA on my machine returns
```
OSA  7C08 ON OSA   7C08 SUBCHANNEL = 0000
OSA  7C09 ON OSA   7C09 SUBCHANNEL = 0001
OSA  7C14 ON OSA   7C14 SUBCHANNEL = 0002
OSA  7C15 ON OSA   7C15 SUBCHANNEL = 0003
```

If you have a guest with certain privileges you may be able to see devices which don't belong to you. To avoid this, add the option V.
e.g.
Q V OSA

Now using the device numbers returned by this command we will
Trace the io starting up on the first device 7c08 & 7c09
In our simplest case we can trace the
start subchannels
like TR SSCH 7C08-7C09
or the halt subchannels
or TR HSCH 7C08-7C09
MSCH's ,STSCH's I think you can guess the rest

Ingo's favourite trick is tracing all the IO's & CCWS & spooling them into the reader of another
VM guest so he can ftp the logfile back to his own machine.I'll do a small bit of this & give you
 a look at the output.

1) Spool stdout to VM reader
SP PRT TO (another vm guest ) or * for the local vm guest
2) Fill the reader with the trace
TR IO 7c08-7c09 INST INT CCW PRT RUN
3) Start up linux
i 00c
4) Finish the trace
TR END
5) close the reader
C PRT

6) list reader contents
RDRLIST
7) copy it to linux4's minidisk
RECEIVE / LOG TXT A1 ( replace
8)
filel & press F11 to look at it
You should see something like:

```
00020942' SSCH  B2334000    0048813C    CC 0    SCH 0000    DEV 7C08
          CPA 000FFDF0   PARM 00E2C9C4    KEY 0  FPI C0  LPM 80
          CCW    000FFDF0  E4200100 00487FE8   0000  E4240100 ........
          IDAL                                      43D8AFE8
          IDAL                                      0FB76000
00020B0A'   I/O DEV 7C08 -> 000197BC'   SCH 0000   PARM 00E2C9C4
00021628' TSCH  B2354000 >> 00488164    CC 0    SCH 0000    DEV 7C08
          CCWA 000FFDF8   DEV STS 0C  SCH STS 00  CNT 00EC
           KEY 0   FPI C0  CC 0   CTLS 4007
00022238' STSCH B2344000 >> 00488108    CC 0    SCH 0000    DEV 7C08
```

If you don't like messing up your readed ( because you possibly booted from it )
you can alternatively spool it to another readers guest.

Other common VM device related commands
---------------------------------------------
These commands are listed only because they have
been of use to me in the past & may be of use to
you too. For more complete info on each of the commands
use type HELP <command> from CMS.
detaching devices
DET <devno range>
ATT <devno range> <guest>
attach a device to guest * for your own guest
READY <devno> cause VM to issue a fake interrupt.

The VARY command is normally only available to VM administrators.
VARY ON PATH <path> TO <devno range>
VARY OFF PATH <PATH> FROM <devno range>
This is used to switch on or off channel paths to devices.

Q CHPID <channel path ID>
This displays state of devices using this channel path
D SCHIB <subchannel>
This displays the subchannel information SCHIB block for the device.
this I believe is also only available to administrators.
DEFINE CTC <devno>
defines a virtual CTC channel to channel connection
2 need to be defined on each guest for the CTC driver to use.
COUPLE  devno userid remote devno
Joins a local virtual device to a remote virtual device
( commonly used for the CTC driver ).

Building a VM ramdisk under CMS which linux can use
def vfb-<blocksize> <subchannel> <number blocks>
blocksize is commonly 4096 for linux.
Formatting it

format <subchannel> <driver letter e.g. x> (blksize <blocksize>

Sharing a disk between multiple guests
LINK userid devno1 devno2 mode password

## GDB on S390

N.B. if compiling for debugging gdb works better without optimisation
( see Compiling programs for debugging )

### invocation

gdb <victim program> <optional corefile>

### Online help

help: gives help on commands
e.g.
help
help display
Note gdb's online help is very good use it.

### Assembly

info registers: displays registers other than floating point.
info all-registers: displays floating points as well.
disassemble: disassembles
e.g.
disassemble without parameters will disassemble the current function
disassemble $pc $pc+10

### Viewing & modifying variables

print or p: displays variable or register
e.g. p/x $sp will display the stack pointer

display: prints variable or register each time program stops
e.g.
display/x $pc will display the program counter
display argc

undisplay : undo's display's

info breakpoints: shows all current breakpoints

info stack: shows stack back trace ( if this doesn't work too well, I'll show you the
stacktrace by hand below ).

info locals: displays local variables.

info args: display current procedure arguments.

set args: will set argc & argv each time the victim program is invoked.

set <variable>=value
set argc=100
set $pc=0

Modifying execution
--------------------
step: steps n lines of sourcecode
step steps 1 line.
step 100 steps 100 lines of code.

next: like step except this will not step into subroutines

stepi: steps a single machine code instruction.
e.g. stepi 100

nexti: steps a single machine code instruction but will not step into subroutines.

finish: will run until exit of the current routine

run: (re)starts a program

cont: continues a program

quit: exits gdb.

breakpoints
------------

break
sets a breakpoint
e.g.

break main

break *$pc

break *0x400618

Here's a really useful one for large programs
rbr
Set a breakpoint for all functions matching REGEXP
e.g.
rbr 390
will set a breakpoint with all functions with 390 in their name.

info breakpoints
lists all breakpoints

delete: delete breakpoint by number or delete them all
e.g.

delete 1 will delete the first breakpoint
delete will delete them all

watch: This will set a watchpoint ( usually hardware assisted ),
This will watch a variable till it changes
e.g.
watch cnt, will watch the variable cnt till it changes.
As an aside unfortunately gdb's, architecture independent watchpoint code
is inconsistent & not very good, watchpoints usually work but not always.

info watchpoints: Display currently active watchpoints

condition: ( another useful one )
Specify breakpoint number N to break only if COND is true.
Usage is `condition N COND', where N is an integer and COND is an
expression to be evaluated whenever breakpoint N is reached.

User defined functions/macros
-----------------------------
define: ( Note this is very very useful,simple & powerful )
usage define <name> <list of commands> end

examples which you should consider putting into .gdbinit in your home directory

```
define d
stepi
disassemble $pc $pc+10
end

define e
nexti
disassemble $pc $pc+10
end
```

Other hard to classify stuff
----------------------------
signal n:
sends the victim program a signal.
e.g. signal 3 will send a SIGQUIT.

info signals:
what gdb does when the victim receives certain signals.

list:
e.g.
list lists current function source
list 1,10 list first 10 lines of current file.
list test.c:1,10

directory:
Adds directories to be searched for source if gdb cannot find the source.
(note it is a bit sensitive about slashes)
e.g. To add the root of the filesystem to the searchpath do

directory //


call <function>
This calls a function in the victim program, this is pretty powerful
e.g.
(gdb) call printf("hello world")
outputs:
$1 = 11

You might now be thinking that the line above didn't work, something extra had to be done.
(gdb) call fflush(stdout)
hello world$2 = 0
As an aside the debugger also calls malloc & free under the hood
to make space for the "hello world" string.

hints
-----
1) command completion works just like bash
( if you are a bad typist like me this really helps )
e.g. hit br <TAB> & cursor up & down :-).

2) if you have a debugging problem that takes a few steps to recreate
put the steps into a file called .gdbinit in your current working directory
if you have defined a few extra useful user defined commands put these in
your home directory & they will be read each time gdb is launched.

A typical .gdbinit file might be.
```
break main
run
break runtime_exception
cont
```

stack chaining in gdb by hand
-----------------------------
This is done using a the same trick described for VM
p/x (*($sp+56))&0x7fffffff get the first backchain.

For z/Architecture
Replace 56 with 112 & ignore the &0x7fffffff
in the macros below & do nasty casts to longs like the following
as gdb unfortunately deals with printed arguments as ints which
messes up everything.
i.e. here is a 3rd backchain dereference
p/x *(long *)(***(long ***)$sp+112)


this outputs
$5 = 0x528f18
on my machine.
Now you can use
info symbol (*($sp+56))&0x7fffffff

you might see something like.
rl_getc + 36 in section .text  telling you what is located at address 0x528f18
Now do.
p/x (*(*$sp+56))&0x7fffffff
This outputs
$6 = 0x528ed0
Now do.
info symbol (*(*$sp+56))&0x7fffffff
rl_read_key + 180 in section .text
now do
p/x (*(**$sp+56))&0x7fffffff
& so on.

Disassembling instructions without debug info
---------------------------------------------
gdb typically complains if there is a lack of debugging
symbols in the disassemble command with
"No function contains specified address." To get around
this do
x/<number lines to disassemble>xi <address>
e.g.
x/20xi 0x400730

Note: Remember gdb has history just like bash you don't need to retype the
whole line just use the up & down arrows.

For more info
-------------
From your linuxbox do
man gdb or info gdb.

core dumps
----------
What a core dump ?,
A core dump is a file generated by the kernel ( if allowed ) which contains the
registers,
& all active pages of the program which has crashed.
From this file gdb will allow you to look at the registers & stack trace &
memory of the
program as if it just crashed on your system, it is usually called core &
created in the
current working directory.
This is very useful in that a customer can mail a core dump to a technical
support department
& the technical support department can reconstruct what happened.
Provided they have an identical copy of this program with debugging symbols
compiled in &
the source base of this build is available.
In short it is far more useful than something like a crash log could ever hope
to be.

In theory all that is missing to restart a core dumped program is a kernel patch

which
will do the following.
1) Make a new kernel task structure
2) Reload all the dumped pages back into the kernel's memory management structures.
3) Do the required clock fixups
4) Get all files & network connections for the process back into an identical state ( really difficult ).
5) A few more difficult things I haven't thought of.

Why have I never seen one ?.
Probably because you haven't used the command
ulimit -c unlimited in bash
to allow core dumps, now do
ulimit -a
to verify that the limit was accepted.

A sample core dump
To create this I'm going to do
ulimit -c unlimited
gdb
to launch gdb (my victim app. ) now be bad & do the following from another telnet/xterm session to the same machine
ps -aux | grep gdb
kill -SIGSEGV <gdb's pid>
or alternatively use killall -SIGSEGV gdb if you have the killall command.
Now look at the core dump.
./gdb core
Displays the following

```
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "s390-ibm-linux"...
Core was generated by `./gdb'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/libncurses.so.4...done.
Reading symbols from /lib/libm.so.6...done.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0  0x40126d1a in read () from /lib/libc.so.6
Setting up the environment for debugging gdb.
Breakpoint 1 at 0x4dc6f8: file utils.c, line 471.
Breakpoint 2 at 0x4d87a4: file top.c, line 2609.
(top-gdb) info stack
#0  0x40126d1a in read () from /lib/libc.so.6
#1  0x528f26 in rl_getc (stream=0x7ffffde8) at input.c:402
#2  0x528ed0 in rl_read_key () at input.c:381
#3  0x5167e6 in readline_internal_char () at readline.c:454
#4  0x5168ee in readline_internal_charloop () at readline.c:507
#5  0x51692c in readline_internal () at readline.c:521
#6  0x5164fe in readline (prompt=0x7ffff810 "\177ÿøx\177ÿ÷Ø\177ÿøxÀ")
```

```
    at readline.c:349
#7  0x4d7a8a in command_line_input (prompt=0x564420 "(gdb) ", repeat=1,
    annotation_suffix=0x4d6b44 "prompt") at top.c:2091
#8  0x4d6cf0 in command_loop () at top.c:1345
#9  0x4e25bc in main (argc=1, argv=0x7ffffdf4) at main.c:635
```

LDD
===
This is a program which lists the shared libraries which a library needs,
Note you also get the relocations of the shared library text segments which
help when using objdump --source.
e.g.

```
 ldd ./gdb
```

outputs

```
libncurses.so.4 => /usr/lib/libncurses.so.4 (0x40018000)
libm.so.6 => /lib/libm.so.6 (0x4005e000)
libc.so.6 => /lib/libc.so.6 (0x40084000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Debugging shared libraries
==========================
Most programs use shared libraries, however it can be very painful
when you single step instruction into a function like printf for the
first time & you end up in functions like _dl_runtime_resolve this is
the ld.so doing lazy binding, lazy binding is a concept in ELF where
shared library functions are not loaded into memory unless they are
actually used, great for saving memory but a pain to debug.
To get around this either relink the program -static or exit gdb type
export LD_BIND_NOW=true this will stop lazy binding & restart the gdb'ing
the program in question.

Debugging modules
=================
As modules are dynamically loaded into the kernel their address can be
anywhere to get around this use the -m option with insmod to emit a load
map which can be piped into a file if required.

The proc file system
====================
What is it ?.
It is a filesystem created by the kernel with files which are created on demand
by the kernel if read, or can be used to modify kernel parameters,
it is a powerful concept.

e.g.

```
cat /proc/sys/net/ipv4/ip_forward
```

On my machine outputs
0
telling me ip_forwarding is not on to switch it on I can do

```
echo 1 >  /proc/sys/net/ipv4/ip_forward
```

cat it again

cat /proc/sys/net/ipv4/ip_forward
On my machine now outputs
1
IP forwarding is on.
There is a lot of useful info in here best found by going in & having a look around,
so I'll take you through some entries I consider important.

All the processes running on the machine have there own entry defined by
/proc/<pid>
So lets have a look at the init process
cd /proc/1

cat cmdline
emits
init [2]

cd /proc/1/fd
This contains numerical entries of all the open files,
some of these you can cat e.g. stdout (2)

cat /proc/29/maps
on my machine emits

```
00400000-00478000 r-xp 00000000 5f:00 4103       /bin/bash
00478000-0047e000 rw-p 00077000 5f:00 4103       /bin/bash
0047e000-00492000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 5f:00 14382      /lib/ld-2.1.2.so
40015000-40016000 rw-p 00014000 5f:00 14382      /lib/ld-2.1.2.so
40016000-40017000 rwxp 00000000 00:00 0
40017000-40018000 rw-p 00000000 00:00 0
40018000-4001b000 r-xp 00000000 5f:00 14435      /lib/libtermcap.so.2.0.8
4001b000-4001c000 rw-p 00002000 5f:00 14435      /lib/libtermcap.so.2.0.8
4001c000-4010d000 r-xp 00000000 5f:00 14387      /lib/libc-2.1.2.so
4010d000-40111000 rw-p 000f0000 5f:00 14387      /lib/libc-2.1.2.so
40111000-40114000 rw-p 00000000 00:00 0
40114000-4011e000 r-xp 00000000 5f:00 14408      /lib/libnss_files-2.1.2.so
4011e000-4011f000 rw-p 00009000 5f:00 14408      /lib/libnss_files-2.1.2.so
7fffd000-80000000 rwxp ffffe000 00:00 0
```

Showing us the shared libraries init uses where they are in memory
& memory access permissions for each virtual memory area.

/proc/1/cwd is a softlink to the current working directory.
/proc/1/root is the root of the filesystem for this process.

/proc/1/mem is the current running processes memory which you
can read & write to like a file.
strace uses this sometimes as it is a bit faster than the
rather inefficient ptrace interface for peeking at DATA.

cat status

Name:   init

```
State:  S (sleeping)
Pid:    1
PPid:   0
Uid:    0       0       0       0
Gid:    0       0       0       0
Groups:
VmSize:      408 kB
VmLck:         0 kB
VmRSS:       208 kB
VmData:       24 kB
VmStk:         8 kB
VmExe:       368 kB
VmLib:         0 kB
SigPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 7fffffffd7f0d8fc
SigCgt: 00000000280b2603
CapInh: 00000000fffffeff
CapPrm: 00000000ffffffff
CapEff: 00000000fffffeff

User PSW:    070de000 80414146
task: 004b6000 tss: 004b62d8 ksp: 004b7ca8 pt_regs: 004b7f68
User GPRS:
00000400  00000000  0000000b  7ffffa90
00000000  00000000  00000000  0045d9f4
0045cafc  7ffffa90  7fffff18  0045cb08
00010400  804039e8  80403af8  7ffff8b0
User ACRS:
00000000  00000000  00000000  00000000
00000001  00000000  00000000  00000000
00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000
Kernel BackChain  CallChain    BackChain  CallChain
       004b7ca8   8002bd0c     004b7d18   8002b92c
       004b7db8   8005cd50     004b7e38   8005d12a
       004b7f08   80019114
```

Showing among other things memory usage & status of some signals &
the processes'es registers from the kernel task_structure
as well as a backchain which may be useful if a process crashes
in the kernel for some unknown reason.

Some driver debugging techniques
================================

debug feature
-------------

Some of our drivers now support a "debug feature" in
/proc/s390dbf see s390dbf.txt in the linux/Documentation directory
for more info.
e.g.
to switch on the lcs "debug feature"
echo 5 > /proc/s390dbf/lcs/level
& then after the error occurred.
cat /proc/s390dbf/lcs/sprintf >/logfile
the logfile now contains some information which may help
tech support resolve a problem in the field.

```
high level debugging network drivers
------------------------------------
ifconfig is a quite useful command
it gives the current state of network drivers.

If you suspect your network device driver is dead
one way to check is type
ifconfig <network device>
e.g. tr0
You should see something like
tr0       Link encap:16/4 Mbps Token Ring (New)  HWaddr 00:04:AC:20:8E:48
          inet addr:9.164.185.132  Bcast:9.164.191.255  Mask:255.255.224.0
          UP BROADCAST RUNNING MULTICAST  MTU:2000  Metric:1
          RX packets:246134 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100

if the device doesn't say up
try
/etc/rc.d/init.d/network start
( this starts the network stack & hopefully calls ifconfig tr0 up ).
ifconfig looks at the output of /proc/net/dev & presents it in a more
presentable form
Now ping the device from a machine in the same subnet.
if the RX packets count & TX packets counts don't increment you probably
have problems.
next
cat /proc/net/arp
Do you see any hardware addresses in the cache if not you may have problems.
Next try
ping -c 5 <broadcast_addr> i.e. the Bcast field above in the output of
ifconfig. Do you see any replies from machines other than the local machine
if not you may have problems. also if the TX packets count in ifconfig
hasn't incremented either you have serious problems in your driver
(e.g. the txbusy field of the network device being stuck on )
or you may have multiple network devices connected.


chandev
-------
There is a new device layer for channel devices, some
drivers e.g. lcs are registered with this layer.
If the device uses the channel device layer you'll be
able to find what interrupts it uses & the current state
of the device.
See the manpage chandev.8 &type cat /proc/chandev for more info.



Starting points for debugging scripting languages etc.
=======================================================

bash/sh
```

bash -x <scriptname>
e.g. bash -x /usr/bin/bashbug
displays the following lines as it executes them.

```
+ MACHINE=i586
+ OS=linux-gnu
+ CC=gcc
+ CFLAGS= -DPROGRAM='bash' -DHOSTTYPE='i586' -DOSTYPE='linux-gnu'
-DMACHTYPE='i586-pc-linux-gnu' -DSHELL -DHAVE_CONFIG_H   -I. -I. -I./lib -O2
-pipe
+ RELEASE=2.01
+ PATCHLEVEL=1
+ RELSTATUS=release
+ MACHTYPE=i586-pc-linux-gnu
```

perl -d <scriptname> runs the perlscript in a fully interactive debugger
<like gdb>.
Type 'h' in the debugger for help.

for debugging java type
jdb <filename> another fully interactive gdb style debugger.
& type ? in the debugger for help.

## Dumptool & Lcrash ( lkcd )

Michael Holzheu & others here at IBM have a fairly mature port of
SGI's lcrash tool which allows one to look at kernel structures in a
running kernel.

It also complements a tool called dumptool which dumps all the kernel's
memory pages & registers to either a tape or a disk.
This can be used by tech support or an ambitious end user do
post mortem debugging of a machine like gdb core dumps.

Going into how to use this tool in detail will be explained
in other documentation supplied by IBM with the patches & the
lcrash homepage http://oss.sgi.com/projects/lkcd/ & the lcrash manpage.

### How they work

Lcrash is a perfectly normal program,however, it requires 2
additional files, Kerntypes which is built using a patch to the
linux kernel sources in the linux root directory & the System.map.

Kerntypes is an objectfile whose sole purpose in life
is to provide stabs debug info to lcrash, to do this
Kerntypes is built from kerntypes.c which just includes the most commonly
referenced header files used when debugging, lcrash can then read the
.stabs section of this file.

Debugging a live system it uses /dev/mem
alternatively for post mortem debugging it uses the data
collected by dumptool.

```
SysRq
=====
This is now supported by linux for s/390 & z/Architecture.
To enable it do compile the kernel with
Kernel Hacking -> Magic SysRq Key Enabled
echo "1" > /proc/sys/kernel/sysrq
also type
echo "8" >/proc/sys/kernel/printk
To make printk output go to console.
On 390 all commands are prefixed with
^-
e.g.
^-t will show tasks.
^-? or some unknown command will display help.
The sysrq key reading is very picky ( I have to type the keys in an
 xterm session & paste them  into the x3270 console )
& it may be wise to predefine the keys as described in the VM hints above

This is particularly useful for syncing disks unmounting & rebooting
if the machine gets partially hung.

Read Documentation/sysrq.txt for more info

References:
===========
Enterprise Systems Architecture Reference Summary
Enterprise Systems Architecture Principles of Operation
Hartmut Penners s390 stack frame sheet.
IBM Mainframe Channel Attachment a technology brief from a CISCO webpage
Various bits of man & info pages of Linux.
Linux & GDB source.
Various info & man pages.
CMS Help on tracing commands.
Linux for s/390 Elf Application Binary Interface
Linux for z/Series Elf Application Binary Interface ( Both Highly Recommended )
z/Architecture Principles of Operation SA22-7832-00
Enterprise Systems Architecture/390 Reference Summary SA22-7209-01 & the
Enterprise Systems Architecture/390 Principles of Operation SA22-7201-05

Special Thanks
==============
Special thanks to Neale Ferguson who maintains a much
prettier HTML version of this page at
http://penguinvm.princeton.edu/notes.html#Debug390
Bob Grainger Stefan Bader & others for reporting bugs
```