Arquitectura de Computadoras
GDB

QUEMU

Para debuguear el TPE con GDB:

- Es necesario indicarle a qemu que en cuanto arranque congele el CPU para que puedan avanzar ustedes manualmente. Opción -S
- También es necesario indicarle a qemu que espere una conexión en gdb. Opción -s
- qemu con "-d int" imprime por pantalla todas las interrupciones, esto es muy útil para hacer debugging
- También permite monitorear muchas otras características con "-monitor stdio". Con esto podemos hacer dump de los registros, ver si esta configuración la memoria dinámica o no, etc.
- Aquí (https://www.qemu.org/docs/master/system/gdb.html) está la parte del manual que explica como configurar las IRQs mientras se utiliza gdb

GDB

GDB necesita la **tabla de símbolos**, de lo contrario solo puede mostrar el assembler. Sin embargo, por más que compilen todo el TP con la opción -g, toda la información de debugging se pierde ya que el formato de salida que le piden al linkear (en los scripts .ld) es binary.

Para esto **pueden linkear 2 veces**, una con output binary y otra con output elf-x86-64 que preserva la información de debugging.

GDB necesita la dirección IP y el puerto donde está escuchando qemu (opción -s de qemu). Por defecto el puerto es 1234 (en realidad la opción -s de qemu es lo mismo que -gdb tcp::1234), luego le indican a GDB estos datos con la opción target remote IP:PORT

El Docker que tienen ya tiene el gdb instalado, y pueden contactar gdb directamente con qemu corriendo en el host.

Extras

run.sh

Sería interesante que el run.sh tome como parámetro si tiene que ejecutar qemu para GDB o no:

```
#!/bin/bash

if [ "$1" = "-d" ]; then
    qemu-system-x86_64 -s -S -hda Image/x64BareBonesImage.qcow2 -m 512 -soundhw pcspk
else
    qemu-system-x86_64 -hda Image/x64BareBonesImage.qcow2 -m 512 -soundhw pcspk
fi
```

GUI para gdb

El dashboard[1] con el tmux (tgdb) podemos usarlo, pero dado los nuevos comandos que hay que agregar, se deberían hacer unos agregados.

1. El script de inicialización del **tgdb**, que por defecto busca el archivo **.tgdbinit** en el home del disco, se debe modificar para que busque primero en la ubicación actual un archivo "customizado", y en caso de no encontrarlo, entonces sí vaya al home a buscar el .tgdbinit por defecto.

```
#!/usr/bin/env bash
base_dir=$(dirname "$0")
ARGS=
if [ -f "./gdbinit" ]; then
      ARGS=$ARGS\ -x\ ./gdbinit
fi

if [ -z "$TMUX" ]
then
    exec_cmd="exec tmux new -n GDB"
else
    exec_cmd="exec tmux new-window -n GDB"
fi

# Agregado para usar .tgdbinit en directorio actual en caso de existir.
# Usado cuando debuguea con docker y configura conexion desde el tgdbinit

if [ -f .tgdbinit ]
then
```

[1] Ver el documento subido al campus en la carpeta: Prácticas / TP3 - Asm y C / gdb.pdf

```
    tgdbinit_path='.tgdbinit'
else
    tgdbinit_path='~/.tgdbinit'
fi

$exec_cmd sh -c "gdb -x $tgdbinit_path $ARGS ${@@Q} ; tmux kill-window -t
\$TMUX_PANE"
```

2. Copiar el archivo **.tgdbinit** a la carpeta donde tienen el contenedor de docker, es decir, donde va a encontrarse su TPE, y modificarlo agregando las siguientes líneas:

```
...
gdb.execute(f'set disassembly-flavor intel')

#PARA MOSTRAR STACK
gdb.execute(f'dashboard memory watch $esp-28 28')
gdb.execute(f'dashboard memory watch $esp 28')

# para debug docker
# set debuginfod enabled off

gdb.execute(f'target remote 192.168.1.109:1234')
gdb.execute(f'add-symbol-file Kernel/kernel.elf 0x100000')
gdb.execute(f'add-symbol-file           Userland/0000-sampleCodeModule.elf
0x400000')

gdb.execute(f'set dir Kernel:Userland:Userland/SampleCodeModule')

gdb.execute(f"dashboard registers -style list 'rax rbx rcx rdx rsi rdi rbp
rsp r8 r9 r10 r11 r12 r13 r14 r15 rip eflags cs ss ds es fs gs fs_base
gs_base k_gs_base cr0 cr2 cr3 cr4 cr8 efer'")

#tmux('select-layout', 'even-horizontal')
#tmux('select-pane', '-t', '0')

end

#dashboard -layout memory source assembly registers stack
dashboard -enabled on
...
```

Obtener IP local

El IP 192.168.1.109 del comando “target remote” en el ejemplo, debe ser su IP local, que lo consiguen de la siguiente manera:

```
apt install net-tools
ifconfig
```

Linkeo a elf64-x86_64

Makefile en el kernel

Por el linkeo en un formato diferente que el explicitado en los scripts .ld, es posible ignorar la opcion del script agregando la opción al comando ld. Si usan gcc también es posible pasar flags directo al linkear, por ejemplo:

Si linkean de esta manera (archivo /Kernel/Makefile):

```
$(LD) $(LDFLAGS) -T kernel.ld -o $(KERNEL) $(LOADEROBJECT) $(OBJECTS) $(OBJECTS_ASM) $(STATICLIBS)
```

Pueden cambiar el formato con:

```
$(LD) $(LDFLAGS) -T kernel.ld --oformat=elf64-x86-64 -o kernel.elf $(LOADEROBJECT) $(OBJECTS) $(OBJECTS_ASM) $(STATICLIBS)
```

Makefile en userland

Si compilan y linkean en un solo paso de esta manera (archivo /Userland/SampleCodeModule/Makefile):

```
$(GCC) $(GCCFLAGS) -I./include -T sampleCodeModule.ld _loader.c $(SOURCES) $(OBJECTS_ASM) -o ../$(MODULE)
```

De hecho gcc main.c compila y linkea.

Pueden cambiar el formato con la opción -Wl que le envía los flags especificados directamente al linker.

```
$(GCC) $(GCCFLAGS) -I./include -T sampleCodeModule.ld -Wl,--oformat=elf64-x86-64 _loader.c $(SOURCES) $(OBJECTS_ASM) -o ../0000-sampleCodeModule.elf
```

Recuerden que **no deben reemplazar los comandos** de su make por estos, **sino agregar** los presentados aqui, ya que el comando original genera el binario que se va a utilizar como siempre y el comando nuevo genera un binario que preserva los símbolos y será usado por gdb únicamente.

Flags "–g" en ambos Makefile.inc

Recuerden también **agregar el flag -g** en los archivos /Kernel/Makefile.inc y /Userland/Makefile.inc

```
GCCFLAGS=-g -m64 -fno-exceptions -fno-asynchronous-unwind-tables -mno-mmx -mno-sse -mno-sse2 -fno-builtin-malloc -fno-builtin-free -fno-builtin-realloc -mno-red-zone -Wall -ffreestanding -nostdlib -fno-common -std=c99
```

Cómo se pierde la info en los distintos formatos:

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_32.html