Debug Batocera

Under construction.

Batocera offers logs which can help explain the more obvious problems, but sometimes you need something a bit more robust to analyze a particular problem, such as if an emulator is taking too long to load something or EmulationStation is incorrectly caching something when it shouldn't.

More extensive debugging tools can be downloaded from http://mirrors.o2switch.fr/batocera/extensions/ and then placed into `/userdata/system` to analyze such problems. On Batocera **v36** and higher, these extensions will be available automatically upon next boot with these files present.

Strace

Strace is a Linux syscall tracer which monitors interactions between programs and the Linux kernel, including system calls, signal deliveries and changes of process states. More extensive documentation can be found on its homepage, but this article will be a crash-course introduction to using it in the context of Batocera.

Launch an application inside of strace

The usual method to use strace to analyze an individual program.

Launch the program as usual with its stderr pointed to a file output (ie. put `2> file.log` at the end of the command) and then add ''strace '' to the start of it.

For instance, if wanting to analyze possible errors that EmulationStation is encountering:

```
strace ./emulationstation 2> strace.log
```

Or if needing to see what an individual emulator is doing:

```
strace /usr/bin/flycast 2> flycast-strace.log
```

It is also possible to just use > to see all output, but that is usually far too overwhelming to be useful.

If you would rather just see the output in the current framebuffer, pipe it through to `tee` or `more` instead:

Fix Me! example here

Attach to an already running process

The `-p` flag can be used to attach strace to a process.

For example, if the application in question has a PID of 26380:

```
strace -p 26380
```

Filtering down the output

The resulting output will show a record of *all* interactions that program did with the system. This is usually overbearing, so it is useful to use `grep` or similar tools to filter down the output to what is relevant to the problem.

For instance, if the problem is related to the opening of a file:

```
cat strace.log | grep "open"
```

which might result in something like:

```
openat(AT_FDCWD, "/media/ssd/dev/batocera-
emulationstation/resources/genres.xml", O_RDONLY) = 6
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libxml2.so.2", O_RDONLY|O_CLOEXEC) =
13
openat(AT_FDCWD, "/media/ssd/dev/batocera-
emulationstation/resources/mamenames.xml", O_RDONLY) = 16
openat(AT_FDCWD, "/media/ssd/dev/batocera-
emulationstation/resources/mamebioses.xml", O_RDONLY) = 16
```

If the problem is not obvious (as is the case many times) then the operation code number can be searched instead:

```
cat strace.log | grep "fd=6"
```

which might result in something like:

```
poll([{fd=6, events=POLLOUT}], 1, 0)    = 1 ([{fd=6, revents=POLLOUT}])
poll([{fd=6, events=POLLIN}], 1, -1)    = 1 ([{fd=6, revents=POLLIN}])
poll([{fd=6, events=POLLOUT}], 1, -1)   = 1 ([{fd=6, revents=POLLOUT}])
poll([{fd=6, events=POLLIN}], 1, -1)    = 1 ([{fd=6, revents=POLLIN}])
```

If looking at a time-sensitive issue (such as an emulator being slow to launch), then

```
strace -c /usr/bin/flycast > /dev/null
```

which might result in something like:

```
% time     seconds  usecs/call     calls    errors syscall
```

```
------ ----------- ----------- --------- --------- ----------------
 89.76    0.008016           4      1912           getdents
  8.71    0.000778           0     11778           lstat
  0.81    0.000072           0      8894           write
  0.60    0.000054           0       943           open
[...]
  0.00    0.000000           0         1           sysinfo
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.008930                 25440         3 total
```

GNU Project Debugger (gdb)

Even more under construction.

Does this require the program to be decompiled on the machine itself?

https://sourceware.org/gdb/

Open the program:

```
gdb /usr/bin/program
```

Then run additional commands in the interactive shell which appears:

```
run
```

Some useful functions:

- `run` → Executes the program from start to end.
- `break` → Sets breakpoint on a particular line.
- `next` → executes next line of code, but don't dive into functions.
- `step` → go to next instruction, diving into the function.
- `continue` → continue normal execution.
- `list` → displays the code.
- `clear` → to clear all breakpoints.
- `quit` → exits out of gdb.

From:
https://wiki.batocera.org/ - **Batocera.linux - Wiki**

Permanent link:
https://wiki.batocera.org/debug_batocera?rev=1666860047

Last update: **2022/10/27 08:40**