RISC-V Support into LLVM's libc: Challenges and Solutions for 32-bit and 64-bit

Mikhail R. Gadelha, PhD

24th October 2024

The LLVM C Library

- LLVM-libc is a libc implementation under the LLVM project umbrella, targeting performance and modularity
- Includes support for various architectures, including x86_64, aarch64, arm, and RISC-V (32 and 64-bit)
- Also supports building for GPU, both AMD and NVIDIA
- Written in modern C++

RISC-V Overview

- An **open** instruction set architecture
- It's not a chip (Core i7)
- It's not a piece of IP (ARM Cortex-M)
- It’s the ISA (x86, AArch64, PowerPC)

Why RISC-V Support in LLVM libc?

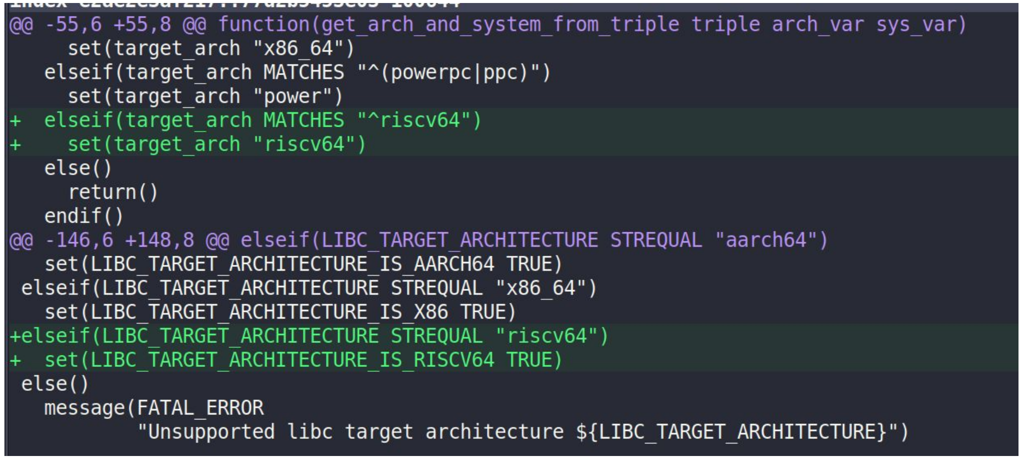
- I wanted to learn more about the architecture 😅
- I was looking for an llvm project to contribute to so libc was the perfect match
- Coincidentally, the first patch with initial RV64 support landed one month prior:
  - Basic implementation of memory functions (memcmp, memmove, memcpy, etc)
  - Partial implementation ctype.h, string.h, math.h and syscalls
  - Missing crt, fenv, threads, longjmp/setjmp, and others

Adding a new arch to LLVM libc

- Basic support can be enabled by changing only 3 files
    - libc/cmake/modules/LLVMLibCArchitectures.cmake: queries arch name (e.g., riscv64)
    - libc/config/linux/**riscv64**/entrypoints.txt: defines the supported functions
    - libc/config/linux/**riscv64**/headers.txt: defines the system headers

Adding a new arch to LLVM libc

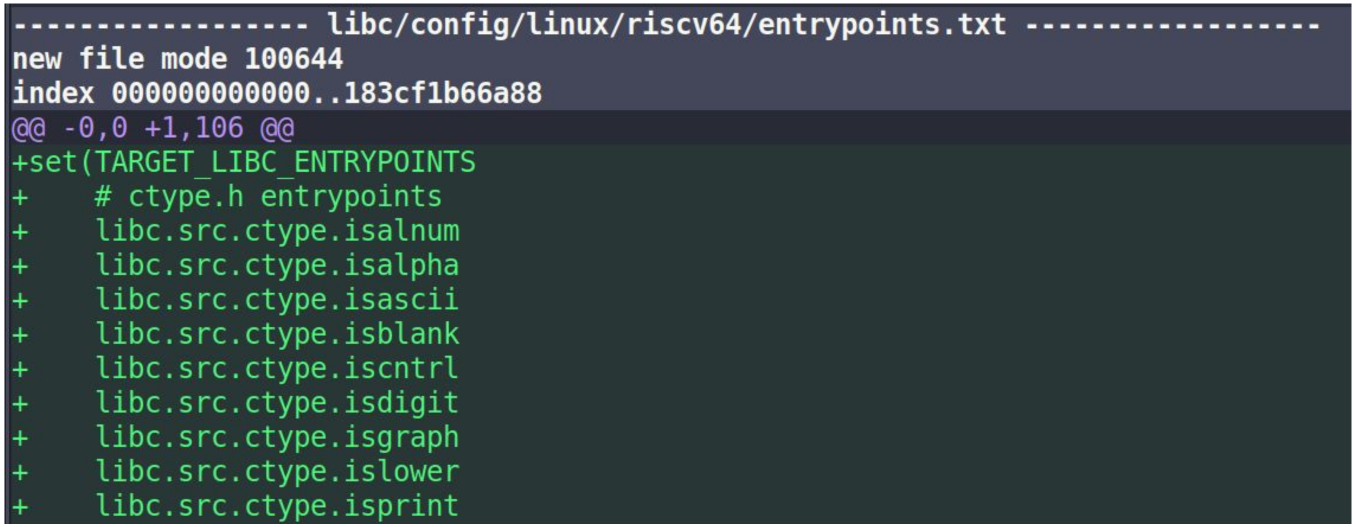
- libc/cmake/modules/LLVMLibCArchitectures.cmake: get the target architecture from the compiler's default target triple



```
@@ -55,6 +55,8 @@ function(get_arch_and_system_from_triple triple arch_var sys_var)
     set(target_arch "x86_64")
   elseif(target_arch MATCHES "^(powerpc|ppc)")
     set(target_arch "power")
+  elseif(target_arch MATCHES "^riscv64")
+    set(target_arch "riscv64")
   else()
     return()
   endif()
@@ -146,6 +148,8 @@ elseif(LIBC_TARGET_ARCHITECTURE STREQUAL "aarch64")
   set(LIBC_TARGET_ARCHITECTURE_IS_AARCH64 TRUE)
 elseif(LIBC_TARGET_ARCHITECTURE STREQUAL "x86_64")
   set(LIBC_TARGET_ARCHITECTURE_IS_X86 TRUE)
+elseif(LIBC_TARGET_ARCHITECTURE STREQUAL "riscv64")
+  set(LIBC_TARGET_ARCHITECTURE_IS_RISCV64 TRUE)
 else()
   message(FATAL_ERROR
           "Unsupported libc target architecture ${LIBC_TARGET_ARCHITECTURE}")
```

Adding a new arch to LLVM libc

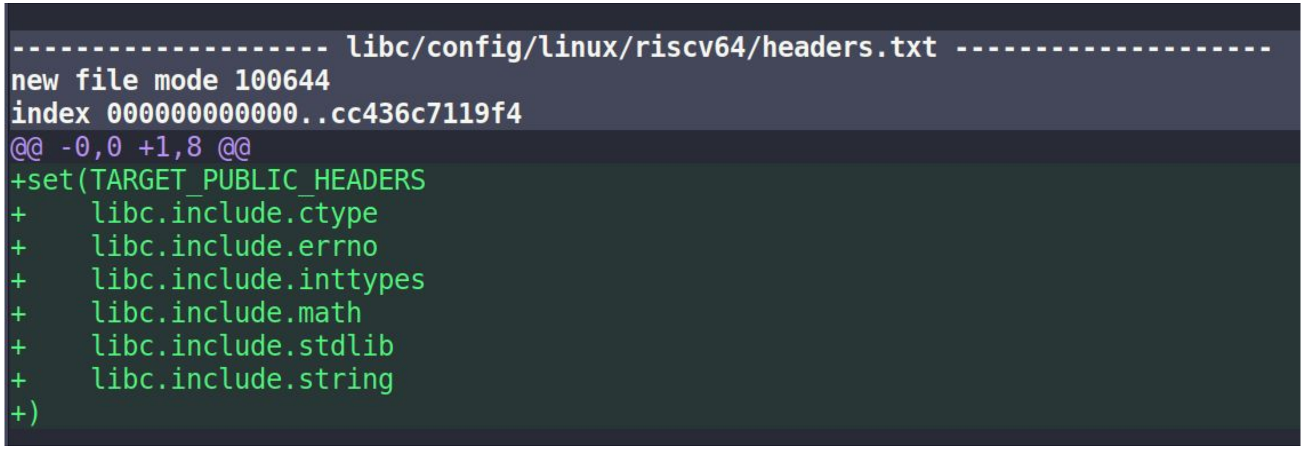
- libc/config/linux/**riscv64**/entrypoints.txt: defines the supported functions



```
------------------ libc/config/linux/riscv64/entrypoints.txt ------------------
new file mode 100644
index 000000000000..183cf1b66a88
@@ -0,0 +1,106 @@
+set(TARGET_LIBC_ENTRYPOINTS
+    # ctype.h entrypoints
+    libc.src.ctype.isalnum
+    libc.src.ctype.isalpha
+    libc.src.ctype.isascii
+    libc.src.ctype.isblank
+    libc.src.ctype.iscntrl
+    libc.src.ctype.isdigit
+    libc.src.ctype.isgraph
+    libc.src.ctype.islower
+    libc.src.ctype.isprint
```

Adding a new arch to LLVM libc

- libc/config/linux/**riscv64**/headers.txt: defines the system headers



```
-------------------- libc/config/linux/riscv64/headers.txt --------------------
new file mode 100644
index 000000000000..cc436c7119f4
@@ -0,0 +1,8 @@
+set(TARGET_PUBLIC_HEADERS
+    libc.include.ctype
+    libc.include.errno
+    libc.include.inttypes
+    libc.include.math
+    libc.include.stdlib
+    libc.include.string
+)
```

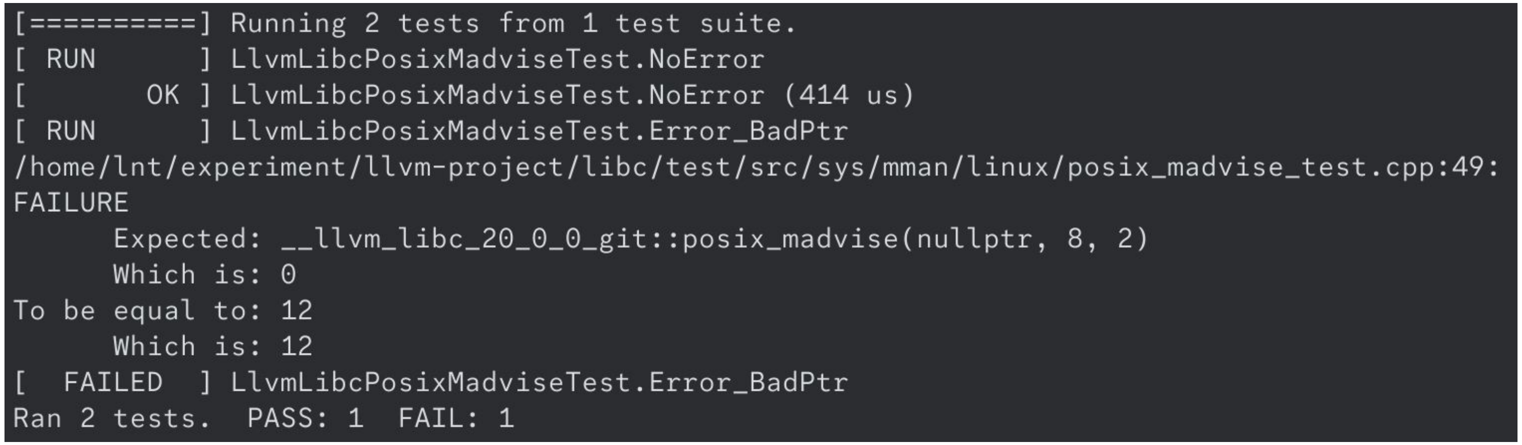
Adding a new arch to LLVM libc

- Once basic support is done, it's a matter of:
  - Adding more functions to entrypoints.txt and headers.txt
  - Run tests
  - Fix bugs
  - Rinse and repeat

Challenges of RISC-V: emulators

- Hardware was not widely available, so we need emulators: qemu, spike, and others
- Initially I used qemu-riscv64 (user space emulator) but several syscalls would fail or succeed when we expected the other way around

Challenges of RISC-V: emulators



```
[==========] Running 2 tests from 1 test suite.
[ RUN      ] LlvmLibcPosixMadviseTest.NoError
[       OK ] LlvmLibcPosixMadviseTest.NoError (414 us)
[ RUN      ] LlvmLibcPosixMadviseTest.Error_BadPtr
/home/lnt/experiment/llvm-project/libc/test/src/sys/mman/linux/posix_madvise_test.cpp:49:
FAILURE
      Expected: __llvm_libc_20_0_0_git::posix_madvise(nullptr, 8, 2)
      Which is: 0
To be equal to: 12
      Which is: 12
[  FAILED  ] LlvmLibcPosixMadviseTest.Error_BadPtr
Ran 2 tests.  PASS: 1  FAIL: 1
```

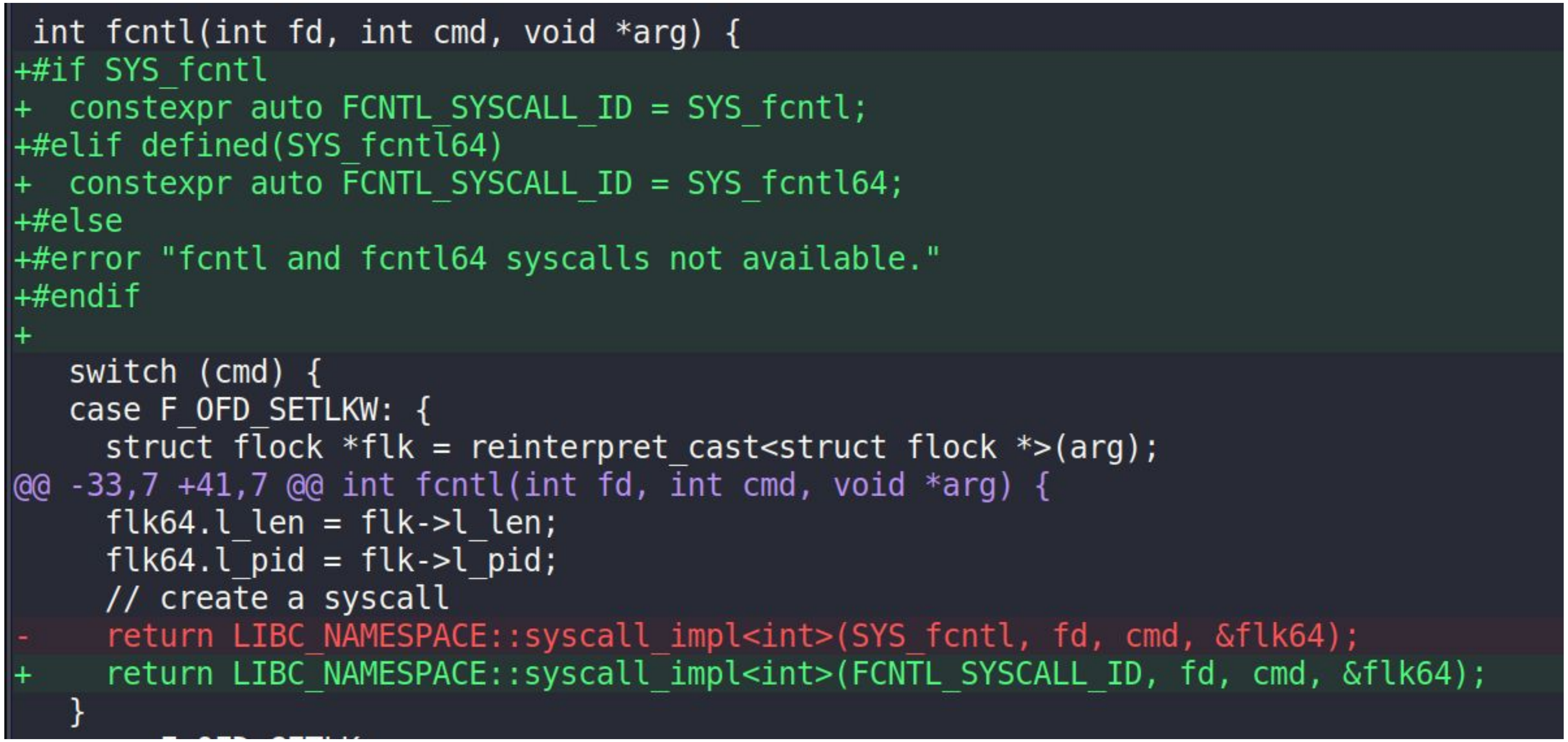
- `posix_madvise`: gives advice about patterns of memory usage
- Returns 0 on success. If the first arg is invalid, returns `ENOMEM (0x12)`

Challenges of RISC-V: emulators

- The solution was to use qemu-system:
  - For RV64 is quite simple, there are images available from several linux distributions
  - For RV32 you need to build your own image, using buildroot or yocto. There is a tutorial on https://discourse.llvm.org/ on how to create it
  - I used yocto since it enables creating an image with gcc/clang as part of the image.
  - For some reason RV32 images are restricted to 1GB of RAM.

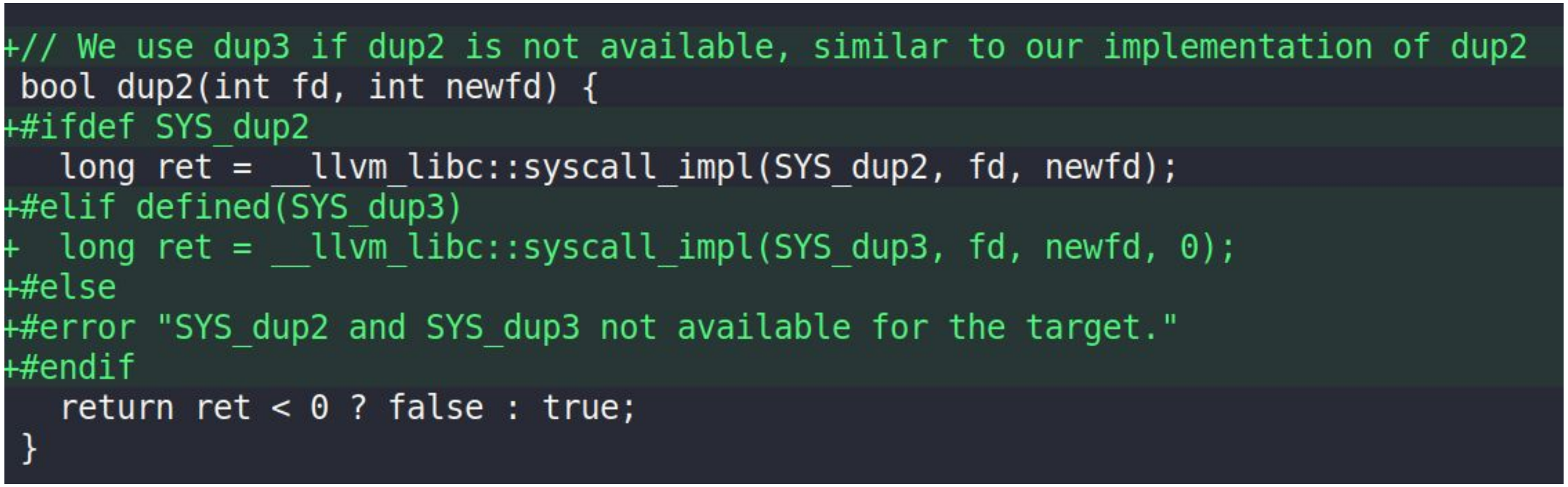
Challenges of RISC-V: syscalls

- Different syscalls compared to other architectures.
- Simplified design compared to x86 or ARM, but lacks backward compatibility for older syscalls, e.g.:
  - `SYS_open → SYS_openat`
  - `SYS_unlink → SYS_unlinkat`
  - `SYS_getdents → SYS_getdents64`
  - `SYS_sched_rr_get_interval → SYS_sched_rr_get_interval_time64`
  - `SYS_wait, SYS_waitpid, SYS_wait3, SYS_wait4 → SYS_waitid`



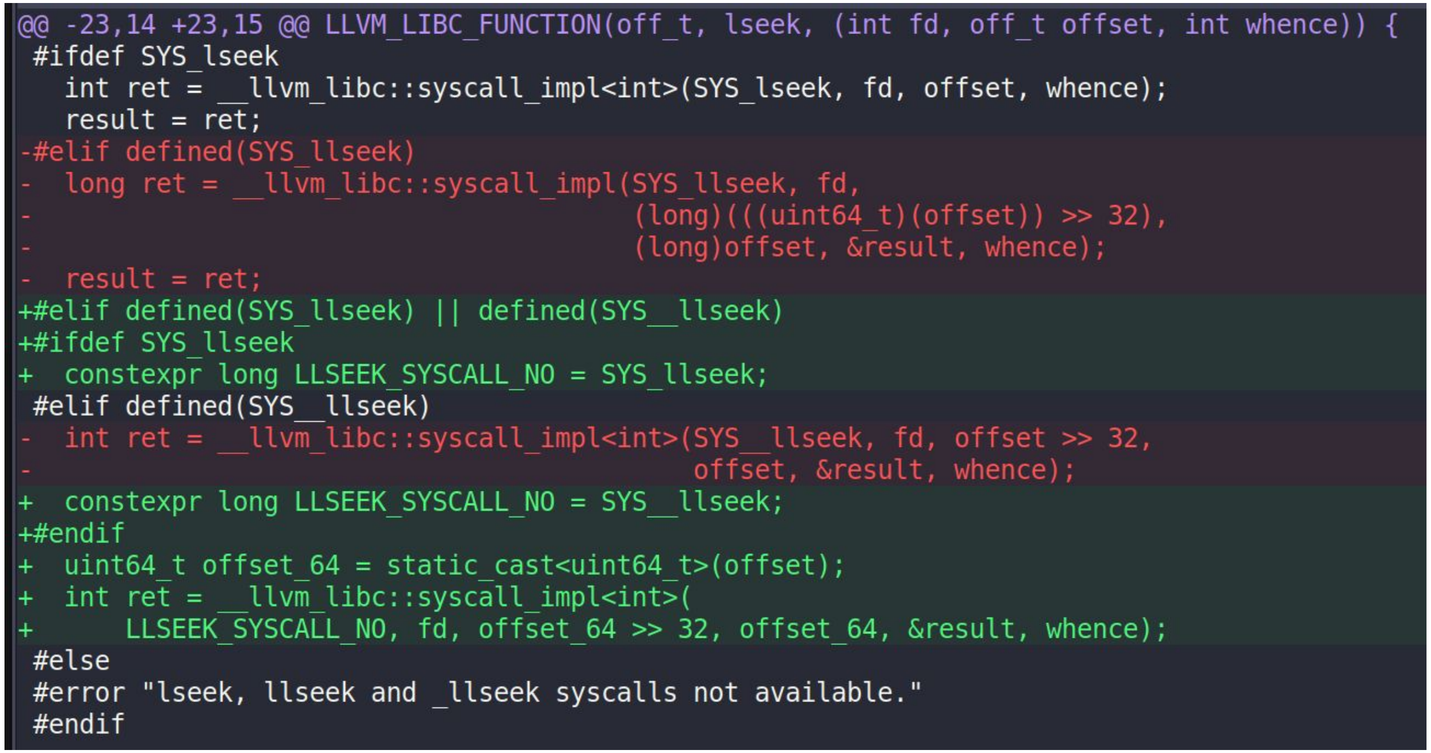
```diff
 int fcntl(int fd, int cmd, void *arg) {
+#if SYS_fcntl
+  constexpr auto FCNTL_SYSCALL_ID = SYS_fcntl;
+#elif defined(SYS_fcntl64)
+  constexpr auto FCNTL_SYSCALL_ID = SYS_fcntl64;
+#else
+#error "fcntl and fcntl64 syscalls not available."
+#endif
+
   switch (cmd) {
   case F_OFD_SETLKW: {
     struct flock *flk = reinterpret_cast<struct flock *>(arg);
@@ -33,7 +41,7 @@ int fcntl(int fd, int cmd, void *arg) {
     flk64.l_len = flk->l_len;
     flk64.l_pid = flk->l_pid;
     // create a syscall
-    return LIBC_NAMESPACE::syscall_impl<int>(SYS_fcntl, fd, cmd, &flk64);
+    return LIBC_NAMESPACE::syscall_impl<int>(FCNTL_SYSCALL_ID, fd, cmd, &flk64);
   }
```

- `fcntl`: manipulate file descriptor



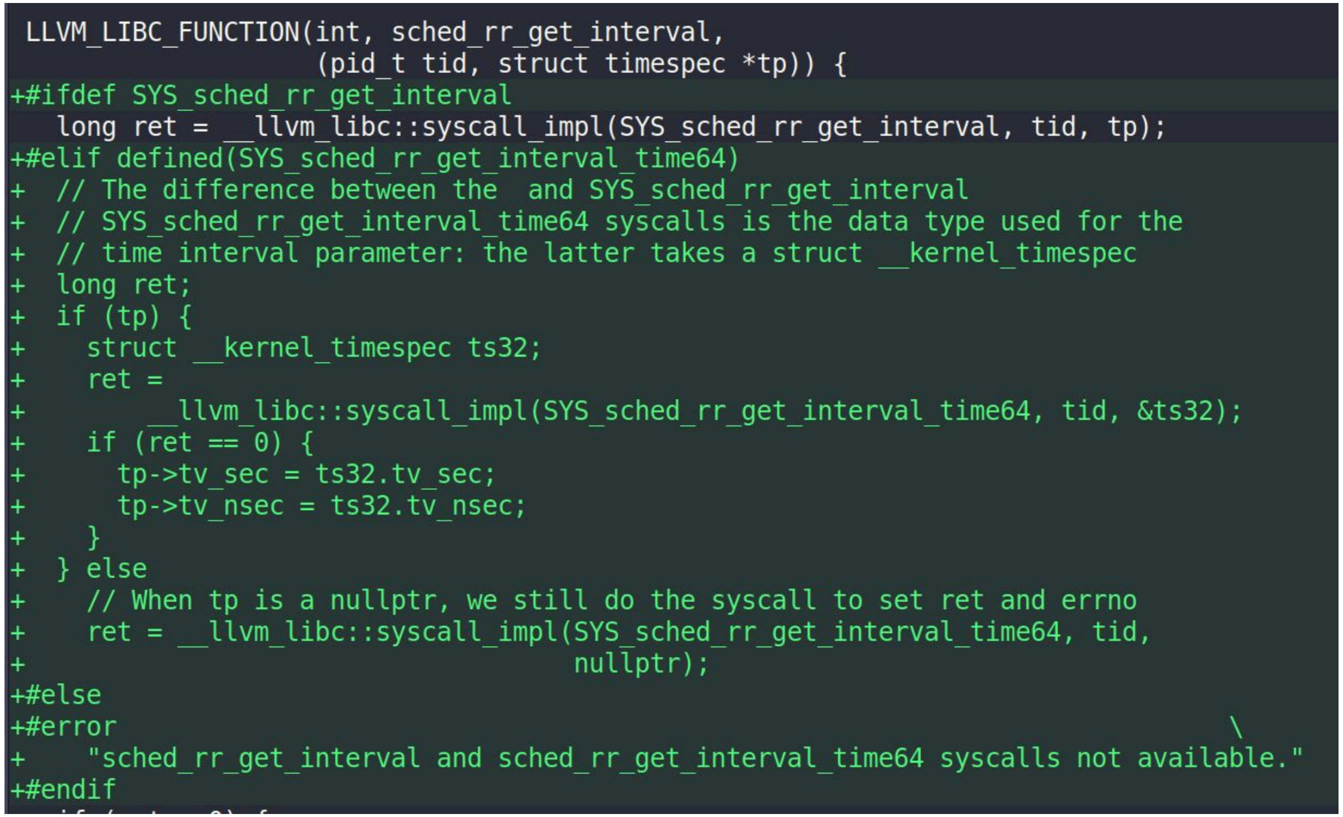
```
+// We use dup3 if dup2 is not available, similar to our implementation of dup2
 bool dup2(int fd, int newfd) {
+#ifdef SYS_dup2
   long ret = __llvm_libc::syscall_impl(SYS_dup2, fd, newfd);
+#elif defined(SYS_dup3)
+  long ret = __llvm_libc::syscall_impl(SYS_dup3, fd, newfd, 0);
+#else
+#error "SYS_dup2 and SYS_dup3 not available for the target."
+#endif
   return ret < 0 ? false : true;
 }
```

- dup, dup2, dup3: duplicate a file descriptor

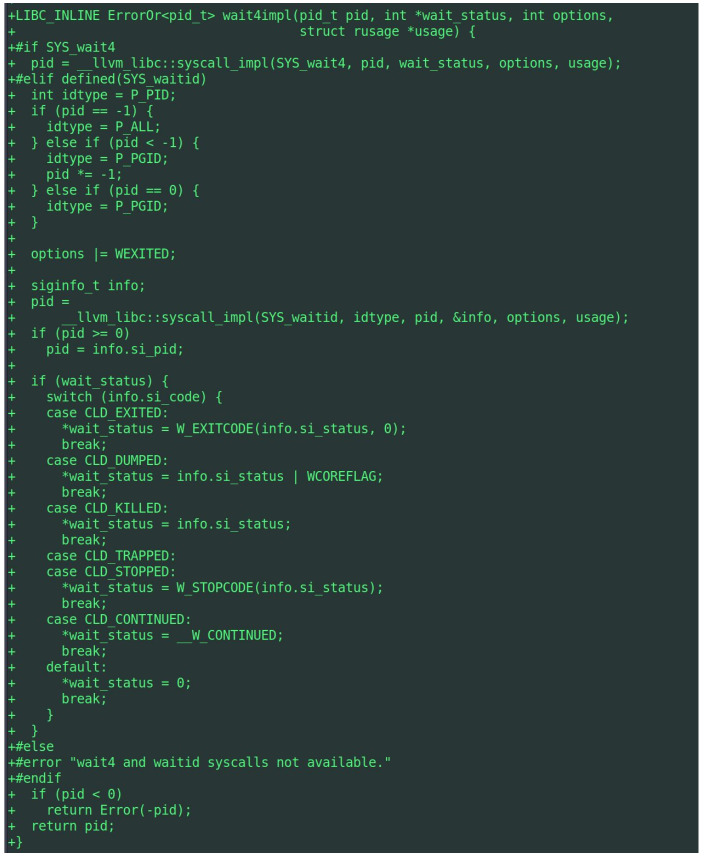


```diff
@@ -23,14 +23,15 @@ LLVM_LIBC_FUNCTION(off_t, lseek, (int fd, off_t offset, int whence)) {
 #ifdef SYS_lseek
   int ret = __llvm_libc::syscall_impl<int>(SYS_lseek, fd, offset, whence);
   result = ret;
-#elif defined(SYS_llseek)
-  long ret = __llvm_libc::syscall_impl(SYS_llseek, fd,
-                                       (long)(((uint64_t)(offset)) >> 32),
-                                       (long)offset, &result, whence);
-  result = ret;
+#elif defined(SYS_llseek) || defined(SYS__llseek)
+#ifdef SYS_llseek
+  constexpr long LLSEEK_SYSCALL_NO = SYS_llseek;
 #elif defined(SYS__llseek)
-  int ret = __llvm_libc::syscall_impl<int>(SYS__llseek, fd, offset >> 32,
-                                           offset, &result, whence);
+  constexpr long LLSEEK_SYSCALL_NO = SYS__llseek;
+#endif
+  uint64_t offset_64 = static_cast<uint64_t>(offset);
+  int ret = __llvm_libc::syscall_impl<int>(
+      LLSEEK_SYSCALL_NO, fd, offset_64 >> 32, offset_64, &result, whence);
 #else
 #error "lseek, llseek and _llseek syscalls not available."
 #endif
```

- `lseek`: reposition read/write file offset

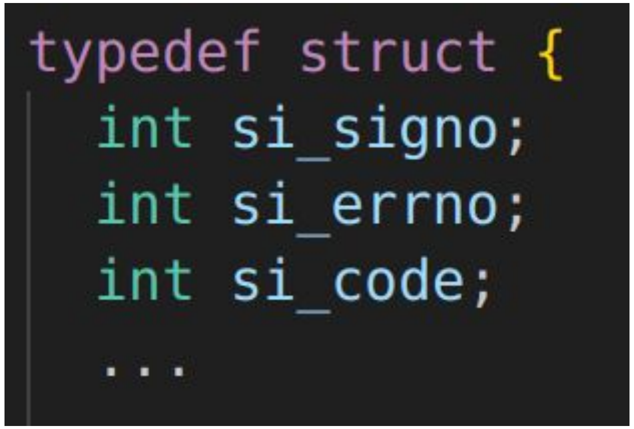

```
 LLVM_LIBC_FUNCTION(int, sched_rr_get_interval,
                    (pid_t tid, struct timespec *tp)) {
+#ifdef SYS_sched_rr_get_interval
   long ret = __llvm_libc::syscall_impl(SYS_sched_rr_get_interval, tid, tp);
+#elif defined(SYS_sched_rr_get_interval_time64)
+  // The difference between the  and SYS_sched_rr_get_interval
+  // SYS_sched_rr_get_interval_time64 syscalls is the data type used for the
+  // time interval parameter: the latter takes a struct __kernel_timespec
+  long ret;
+  if (tp) {
+    struct __kernel_timespec ts32;
+    ret =
+        __llvm_libc::syscall_impl(SYS_sched_rr_get_interval_time64, tid, &ts32);
+    if (ret == 0) {
+      tp->tv_sec = ts32.tv_sec;
+      tp->tv_nsec = ts32.tv_nsec;
+    }
+  } else
+    // When tp is a nullptr, we still do the syscall to set ret and errno
+    ret = __llvm_libc::syscall_impl(SYS_sched_rr_get_interval_time64, tid,
+                                    nullptr);
+#else
+#error                                                                    \
+    "sched_rr_get_interval and sched_rr_get_interval_time64 syscalls not available."
+#endif
```

- `sched_rr_get_interval`: get the SCHED_RR interval for the named process



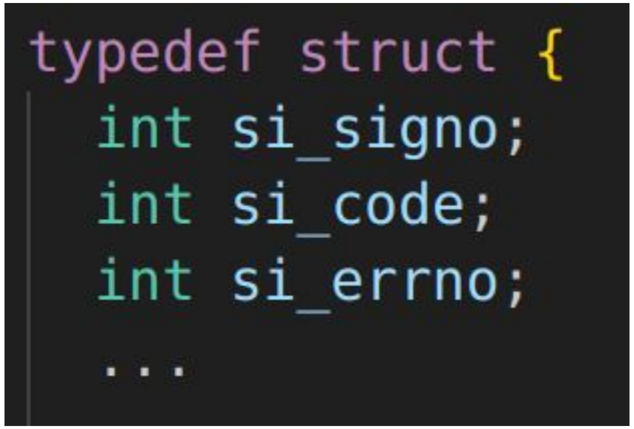
```
+LIBC_INLINE ErrorOr<pid_t> wait4impl(pid_t pid, int *wait_status, int options,
+                                      struct rusage *usage) {
+#if SYS_wait4
+  pid = __llvm_libc::syscall_impl(SYS_wait4, pid, wait_status, options, usage);
+#elif defined(SYS_waitid)
+  int idtype = P_PID;
+  if (pid == -1) {
+    idtype = P_ALL;
+  } else if (pid < -1) {
+    idtype = P_PGID;
+    pid *= -1;
+  } else if (pid == 0) {
+    idtype = P_PGID;
+  }
+
+  options |= WEXITED;
+
+  siginfo_t info;
+  pid =
+      __llvm_libc::syscall_impl(SYS_waitid, idtype, pid, &info, options, usage);
+  if (pid >= 0)
+    pid = info.si_pid;
+
+  if (wait_status) {
+    switch (info.si_code) {
+    case CLD_EXITED:
+      *wait_status = W_EXITCODE(info.si_status, 0);
+      break;
+    case CLD_DUMPED:
+      *wait_status = info.si_status | WCOREFLAG;
+      break;
+    case CLD_KILLED:
+      *wait_status = info.si_status;
+      break;
+    case CLD_TRAPPED:
+    case CLD_STOPPED:
+      *wait_status = W_STOPCODE(info.si_status);
+      break;
+    case CLD_CONTINUED:
+      *wait_status = __W_CONTINUED;
+      break;
+    default:
+      *wait_status = 0;
+      break;
+    }
+  }
+#else
+#error "wait4 and waitid syscalls not available."
+#endif
+  if (pid < 0)
+    return Error(-pid);
+  return pid;
+}
```

- This is `SYS_waitid` that handles `SYS_wait`, `SYS_waitpid`, `SYS_wait3`, `SYS_wait4`



```
typedef struct {
  int si_signo;
  int si_errno;
  int si_code;
  ...
```

x86, arm, riscv



```
typedef struct {
  int si_signo;
  int si_code;
  int si_errno;
  ...
```

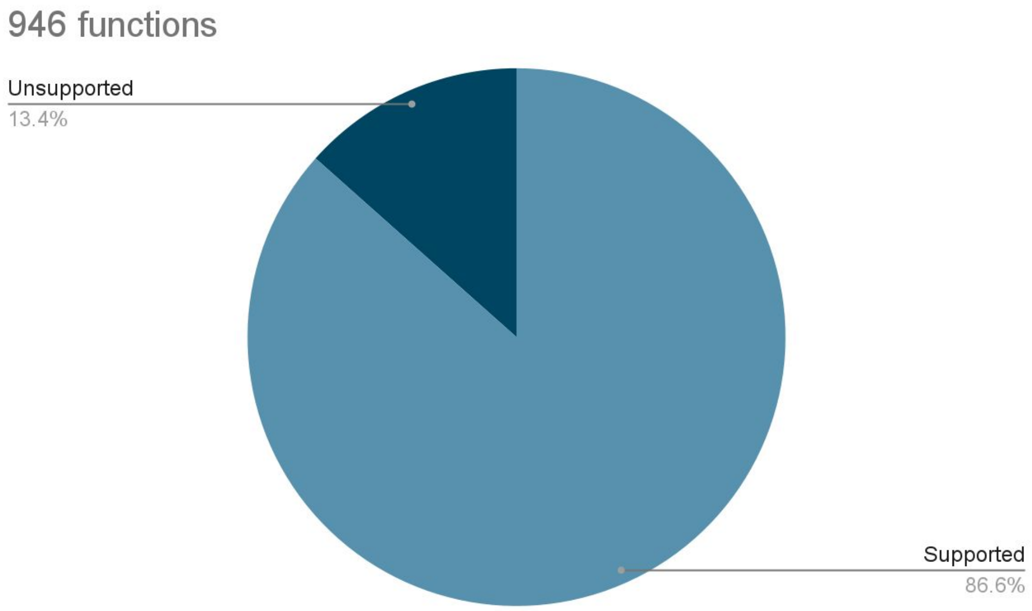
MIPS

- `struct siginfo_t`: data structure containing signal information; it is passed as the second parameter to a user signal handler function.

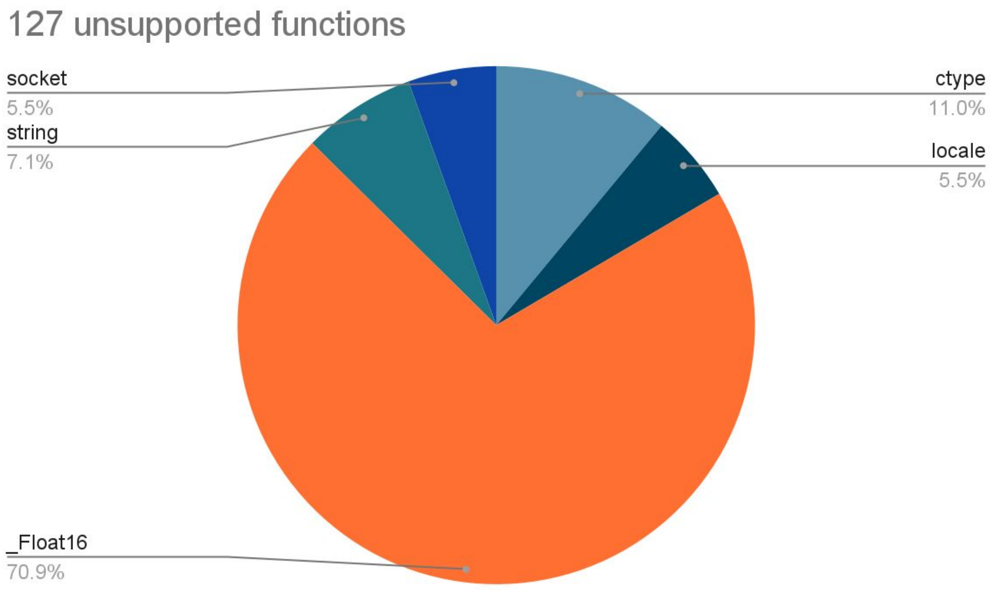
Challenges of RISC-V: others

- Some tests can't fail because of different syscalls and need to be disabled, e.g., `SYS_epoll_create`
- `rand` was using xorshift64star pseudo random number generator, but the LSB was not uniform enough when executed in 32-bit systems
- implicit conversions, e.g., `ssize_t size = sizeof(AuxEntry);`
- RV32 has 128-bit `long double` but no `int128_t`.

RISC-V on libC Today



RISC-V on libC Today



- _Float16: https://github.com/llvm/llvm-project/issues/107607

Thank you!

igalia