# 高清的 KernelX

## 简介

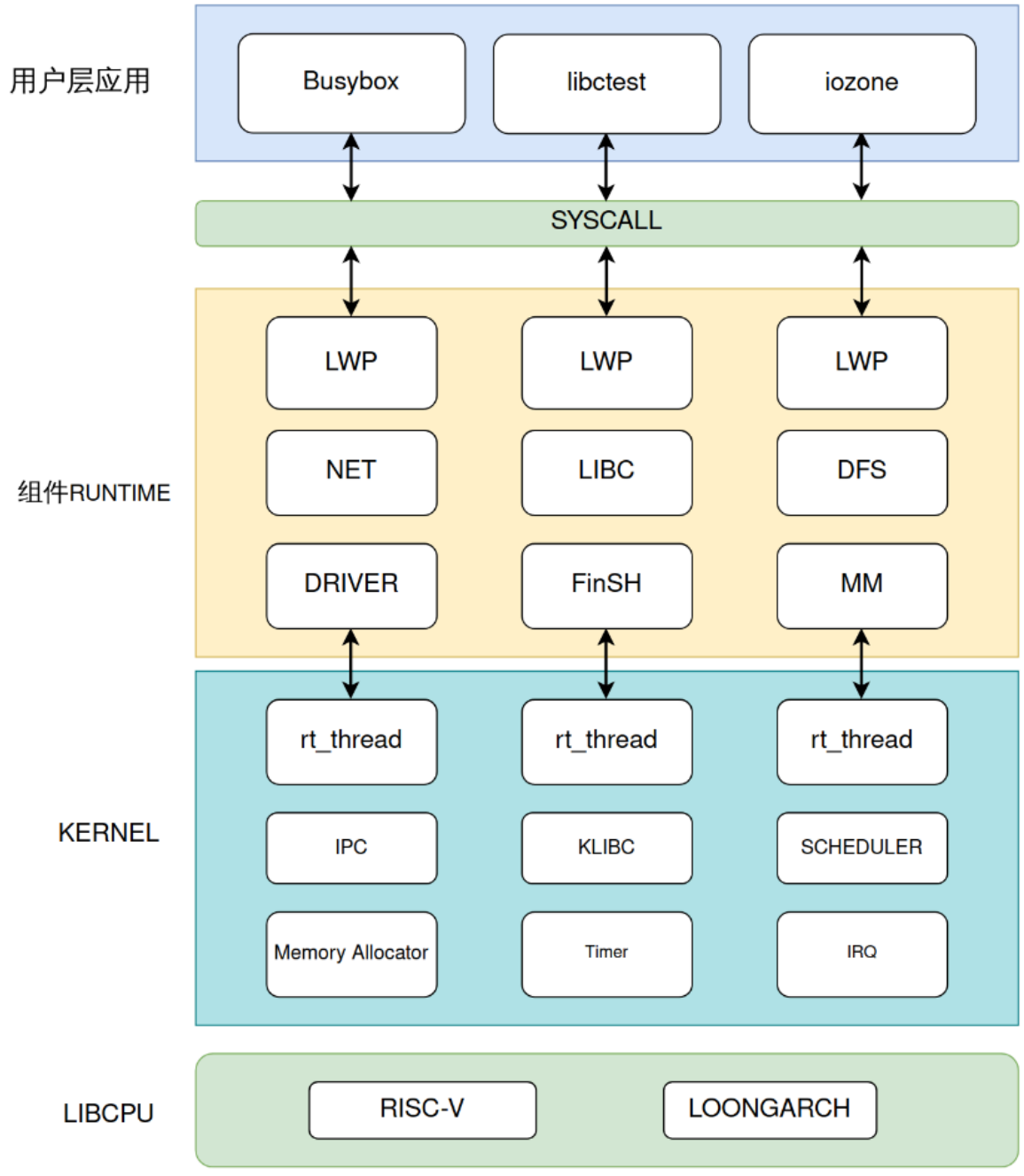
KernelX 是一个基于 RT-Thread Smart 的，使用 c 开发的微内核操作系统。

参赛队员为来自杭州电子科技大学的 张逸轩 刘镇睿 丁宏阳

由于 Gitlab 仓库大小限制，我们将带有提交记录的仓库上传至 GitHub: KernelX

同时，我们给出了对应的 PPT 和 视频：网盘链接

## 架构



# 关于 RT-Thread

RT-Thread(Real Time-Thread), 是一款广泛运用于嵌入式的实时多线程操作系统。RT-Thread 主要使用 C 语言编写，参考了面向对象设计的设计范式。同时，RT-Thread 采取的是微内核架构，具有一个极简的内核以及丰富的拓展、组件，同时支持在线软件包管理，提供更加丰富的功能和强大的裁剪能力以适应不同的设备。

# 设计理念

我们选择以工业界广泛应用的 RT-Thread 作为基础进行拓展和创新，构建了 KernelX 微内核操作系统。我们基于以下理念：首先，我们希望立足于实际的生产，开发一款可以在实际应用场景中部署的操作系统，所以我们选择了在嵌入式领域被诸多使用的 RT-Thread。于此同时，RT-Thread 作为成熟的实时操作系统提供了坚实的微内核架构基础，使我们能够专注于系统功能的完善与优化。我们也希望借此机会深入理解微内核的设计理念和设计哲学。基于种种原因，我们选择 RT-Thread, 并且为他做了修复、优化、功能实现和设计兼容。

# 文档

我们将文档划分为：

总文档
环境文档
RT-Thread文档
组件文档

# 我们的工作

## 修复RT-Thread原有函数实现问题

修复RT-Thread中的一些函数实现问题，例如dfs模块下的`openat`函数，在得到了`dirfd`的绝对路径之后，会直接打开`dirfd`的目录，不拼接相对路径，导致了运行出错。

原有代码（位于components/dfs/src/dfs_posix.c）：

```
int openat(int dirfd, const char *path, int flag, ...)
{
    struct dfs_file *d;
    char *fullpath;
    int fd;

    if (!path)
    {
        rt_set_errno(-EBADF);
        return -1;
    }

    fullpath = (char*)path;

    if (path[0] != '/')
    {
        if (dirfd != AT_FDCWD)
        {
            d = fd_get(dirfd);
            if (!d || !d->vnode)
```

```
            {
                rt_set_errno(-EBADF);
                return -1;
            }

            fullpath = dfs_dentry_full_path(d->dentry);
            if (!fullpath)
            {
                rt_set_errno(-ENOMEM);
                return -1;
            }
        }
    }

    fd = open(fullpath, flag, 0);

    if (fullpath != path)
    {
        rt_free(fullpath);
    }

    return fd;
}
```

我们的修改:

```
if (path[0] != '/') {
    if (dirfd != AT_FDCWD)
    {
        d = fd_get(dirfd);
        if (!d || !d->vnode)
        {
            rt_set_errno(-EBADF);
            return -1;
        }

        char *dirpath = dfs_dentry_full_path(d->dentry);
        size_t dirpath_len = strlen(dirpath);
        size_t path_len = strlen(path);
        fullpath = (char *)rt_malloc(dirpath_len + 1 + path_len + 1);
        rt_strcpy(fullpath, dirpath);
        fullpath[dirpath_len] = '/';
        rt_strcpy(fullpath + dirpath_len + 1, path);
        fullpath[dirpath_len + 1 + path_len] = '\0';
        rt_free(dirpath);

        // fullpath = dfs_dentry_full_path(d->dentry);
        if (!fullpath)
        {
            rt_set_errno(-ENOMEM);
            return -1;
        }
    }
}
```

对于我们发现的这类问题，我们计划在进一步测试之后，向RT-Thread项目提交PR。

## 修改系统调用格式和Linux一致

将RT-Thread中LWP实现的系统调用修改为和Linux一致。虽然RT-Thread已经实现了一些系统调用，但是他们的参数传递方式和Linux有出入。例如`clone`函数的实现，RT-Thread的原有实现将`clone`和`fork`的实现分开，`clone`只负责产生线程，`fork`负责产生进程，同时，`clone`使用一个`void *`来传递六个参数，而Linux则是直接使用寄存器传递六个参数，我们按照原有的逻辑，重写了`clone`系统调用，合并了`clone`和`fork`，并将传参方式改为了直接参数传递。这样的系统调用还有很多,例如`brk`的系统逻辑。系统调用号也需要进行修改。

原有的`clone`和`fork`函数声明：

```
long _sys_clone(void *arg[]);
sysret_t sys_fork(void);
```

修改之后和Linux一致：

```
sysret_t syscall_clone(unsigned long flags, void *user_stack, int *new_tid, void
*tls, int *clear_tid);
```

RT-Thread原有的`brk`逻辑会自动将堆顶向上对齐到页边界，并且暴露给用户接口，但是Linux中往往是用户申请堆顶为多少，就把堆顶设置为多少返回给用户，这导致了测试无法通过。我们加入了一个新的变量来记录用户申请到的堆顶，如果几次申请的新堆顶都再一个页内，我们就无需再为这个用户程序的堆分配页。

RT-Thread使用一个函数数组来记录系统调用号和系统调用操作函数的对应关系，但是这个对应关系和Linux的规定有所区别，我们根据具体的架构，修改了系统调用号使之与Linux一致。

## 增加系统调用

RT-Thread虽然提供了一些系统调用的实现，但是这些系统调用并不足以支持测例的运行。我们通过运用RT-Thread提供的运行时环境，增加一些系统调用，例如：

- `fnctl`
- `writev`
- `sendfile`
- `fstatat`
- `readv`
- `get_euid`
- `times`
- `mprotect`
- `membarrier`
- `sync`
- `fsnyc`
- `readlinkat`
- `getrusage`

- `umask`

等等，同时我们优化了代码结构，提升了代码的可读性。

详细的文档请阅读 qemu-edu/docs 下的文档

# 快速启动

如果在非评测机下的linux环境，我们提供了 docker 环境。

我们提供了一个 python 脚本用来快速构建、启动镜像。

```
python3 run.py # 这一步会自动检索工具链并且安装，自动生成 docker 镜像并且启动，进入。

# 现在我们的目录是 /code

cd ./oscomp/rv
make all # 编译测试环境

cd /code/machines/qemu-virt-riscv64
pkgs --update
scons -c
scons -j12 # 编译系统

./run.sh ../../testsuits-for-oskernel/releases/sdcard-rv.img # 启动系统
```

## 在评测机环境

首先我们需要下载必要的工具，包括scons构建工具和一些python3必要的库

```
apt-get -y update
apt-get -y install scons python3-kconfiglib python3-tqdm python3-requests
python3-yaml
```

然后下载gcc交叉编译工具链并安装opt目录下，同时设置环境变量

```
wget --no-check-certificate https://download.rt-thread.org/download/rt-
smart/toolchains/riscv64gc-linux-musleabi_for_x86_64-pc-linux-gnu_latest.tar.bz2
tar jxvf /root/toolchains/qemu-virt-riscv64/riscv64gc-linux-musleabi_for_x86_64-
pc-linux-gnu_latest.tar.bz2 -C /opt

bash ./toolchains/install_ubuntu.sh --gitee
source ~/.env/env.sh

export PATH=/opt/riscv64gc-linux-musleabi/bin:$PATH
```

到指定目录下构建

```
cd ./machines/qemu-virt-riscv64
pkgs --update
scons -j$(nproc)
```

同时，我们需要一个存放自己测试脚本的磁盘，在启动的时候OS会自动挂载

```
cd ./oscomp/rv
make all
```

磁盘位于./oscomp/rv/build/disk.img。

详细的环境逻辑请参考文档：KernelX-环境

# 项目结构

```
.
├── compile_commands.json
├── Containerfile // Docker 生成配置
├── docs // 文档
│   ├── components // 组件文档目录
│   ├── img
│   ├── KernelX-环境.md
│   ├── KernelX-介绍.md
│   └── RT-Thread-介绍.md
├── gen_bear.sh // 用于 bear 生成 compile_commands 的脚本
├── LICENSE
├── machines // 板级支持
│   ├── qemu-loongarch
│   ├── qemu-virt-riscv64
├── Makefile // makefile, 用于测试
├── oscomp // 测试目录, 用来生成供测试的文件
├── README.md
├── rt-thread // RT-Thread 目录, 也是我们OS的主目录
│   ├── bsp //同样是板级支持
│   ├── ChangeLog.md
│   ├── components // 组件目录, 下面包括多种组件
│   ├── examples
│   ├── include
│   ├── Kconfig
│   ├── libcpu // 提供不同架构的支持
│   ├── LICENSE
│   ├── README_de.md
│   ├── README_es.md
│   ├── README.md
│   ├── README_zh.md
│   ├── src // 源代码, 也就是内核代码
│   └── tools
├── run.py // 我们的用于在普通 linux 环境下启动环境的脚本
├── testsuits-for-oskernel // 官方测试的克隆, 同时修改了 Makefile 以提升编译效率
└── toolchains // 我们需要的工具链, 一般是在 run.py 里面下载
    ├── install_ubuntu.sh
    ├── qemu-longarch
    └── qemu-virt-riscv64
```

# 开源引用声明

KernelX 是在 RT-Thread 的基础上进行开发的，项目路径： RT-Thread