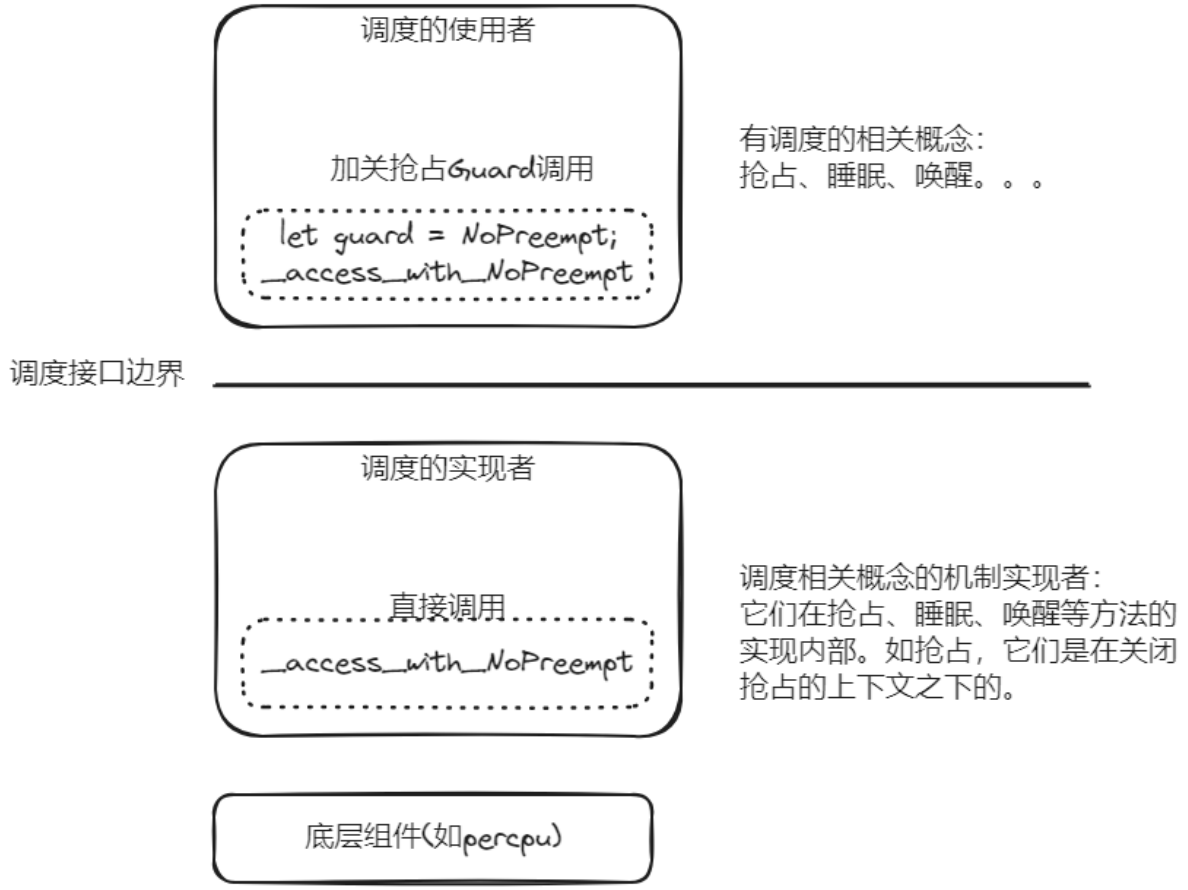
单向依赖问题1：调整拆分与关闭抢占相关的那个kernel_guard，并处理与之相关的percpu。

完成。这里面的复杂问题是，如percpu之类的组件与调度相关组件之间有一个环的关系。percpu被可能被调度相关的组件调用，而它的原始实现中又反过来会调用调度的方法(例如关闭抢占和开启抢占时的唤醒)。

目前的处理方式：把percpu的涉及调度的方法提出去，放在上层处理。

如此的理由如下图：



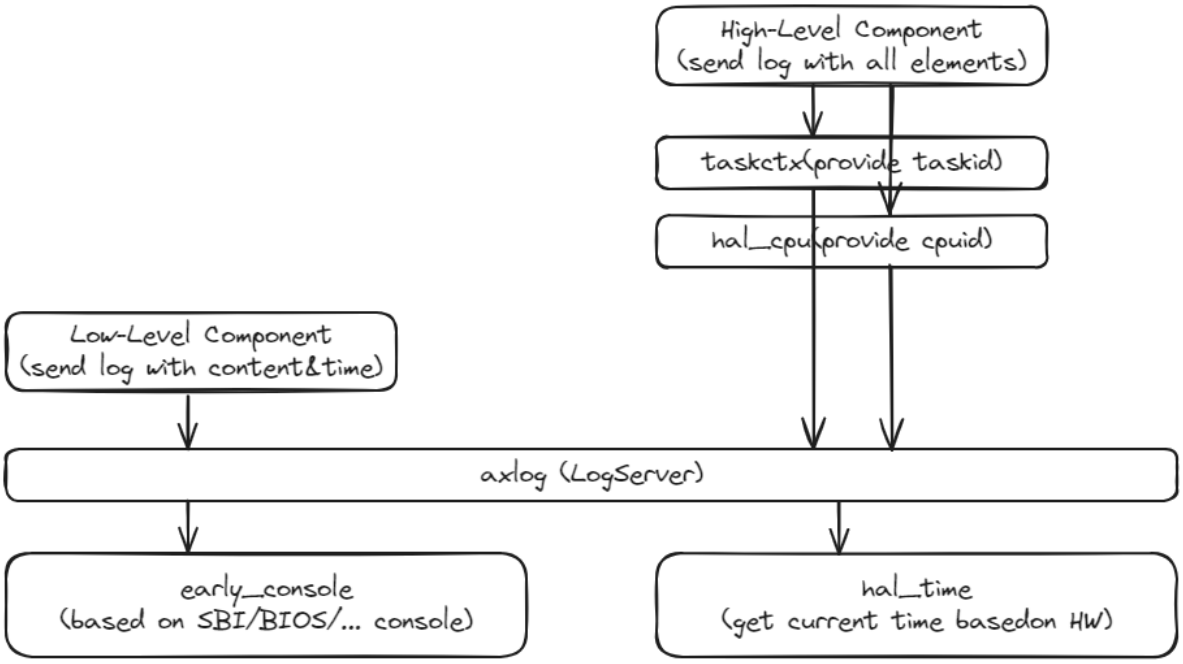
参考的思路来自Linux Kernel，它的调度核心方法schedule就是先关抢占、再关中断，之后的各种内部对runq之类设施的调用都是在关抢占、关中断的上下文之下的，所以调用的都是XXX_locked()和XXX_noPreempt()的函数版本，参见其注释。以上是我的理解。

单向依赖问题2：处理axlog中的crate_interface依赖。

完成。原来的axlog是一个C/S模式，涉及四个要素：日志内容的输出，时间获取、CPU ID和TaskID。其中前两个是必须的，后两个可选。

由于组件层次位置的关系，各个组件都可以输出前两个，但是部分层次低的组件取不到后面两个(他们的位置比TaskCtx和HAL_CPU的位置还低)。

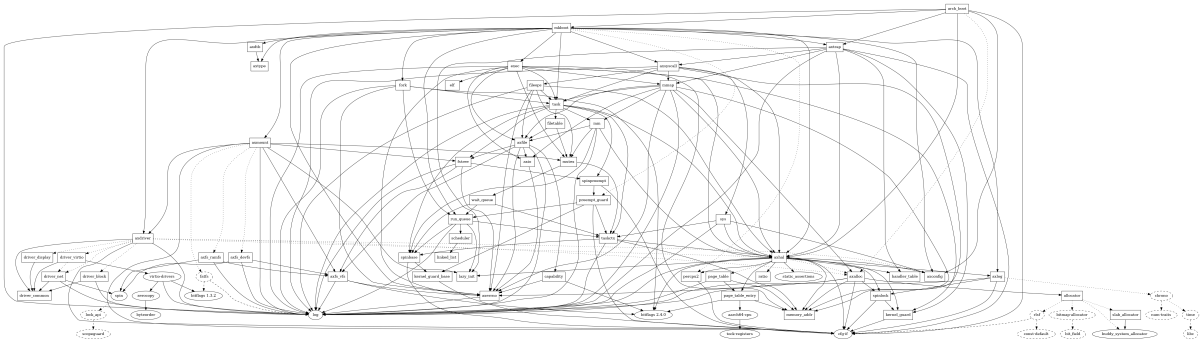
其实axlog的LogServer本身已经部分考虑了对后两个要素的缺省处理，最后想达成的方案：



目前，按照上面思路采取最简单的处理方式，都只是发送了content和time，可以满足记日志的基本要求。

下步要通过macro等方式让上下层分别发不同数量的信息。

目前达到的组件层次状态



LTP syscall测试的进展

仍是针对测试框架本身涉及syscall的支持，新增和完善部分syscall实现，目前大约涉及35个左右。

```
//
// Linux syscall
//
const LINUX_SYSCALL_GETCWD:     usize = 0x11;
const LINUX_SYSCALL_IOCTL:      usize = 0x1d;
const LINUX_SYSCALL_MKDIRAT:    usize = 0x22;
const LINUX_SYSCALL_UNLINKAT:   usize = 0x23;
const LINUX_SYSCALL_FACCESSAT:  usize = 0x30;
const LINUX_SYSCALL_CHDIR:      usize = 0x31;
const LINUX_SYSCALL_CHMODAT:    usize = 0x35;
const LINUX_SYSCALL_CHOWNAT:    usize = 0x36;
const LINUX_SYSCALL_OPENAT:     usize = 0x38;
const LINUX_SYSCALL_CLOSE:      usize = 0x39;
const LINUX_SYSCALL_GETDENTS64: usize = 0x3d;
```

```
const LINUX_SYSCALL_READ:        usize = 0x3f;
const LINUX_SYSCALL_WRITE:       usize = 0x40;
const LINUX_SYSCALL_WRITEV:      usize = 0x42;
const LINUX_SYSCALL_READLINKAT: usize = 0x4e;
const LINUX_SYSCALL_FSTATAT:     usize = 0x4f;
const LINUX_SYSCALL_EXIT:        usize = 0x5d;
const LINUX_SYSCALL_EXIT_GROUP: usize = 0x5e;
const LINUX_SYSCALL_TGKILL:      usize = 0x83;
const LINUX_SYSCALL_UNAME:       usize = 0xa0;
const LINUX_SYSCALL_GETPID:      usize = 0xac;
const LINUX_SYSCALL_GETGID:      usize = 0xb0;
const LINUX_SYSCALL_GETTID:      usize = 0xb2;
const LINUX_SYSCALL_BRK:         usize = 0xd6;
const LINUX_SYSCALL_MUNMAP:      usize = 0xd7;
const LINUX_SYSCALL_MMAP:        usize = 0xde;
const LINUX_SYSCALL_MPROTECT:    usize = 0xe2;
const LINUX_SYSCALL_MSYNC:       usize = 0xe3;
const LINUX_SYSCALL_PRLIMIT64:   usize = 0x105;
const LINUX_SYSCALL_GETRANDOM:   usize = 0x116;

const LINUX_SYSCALL_SET_TID_ADDRESS:     usize = 0x60;
const LINUX_SYSCALL_SET_ROBUST_LIST:     usize = 0x63;
const LINUX_SYSCALL_CLOCK_GETTIME:       usize = 0x71;
const LINUX_SYSCALL_RT_SIGACTION:        usize = 0x86;
const LINUX_SYSCALL_RT_SIGPROCMASK:      usize = 0x87;
```

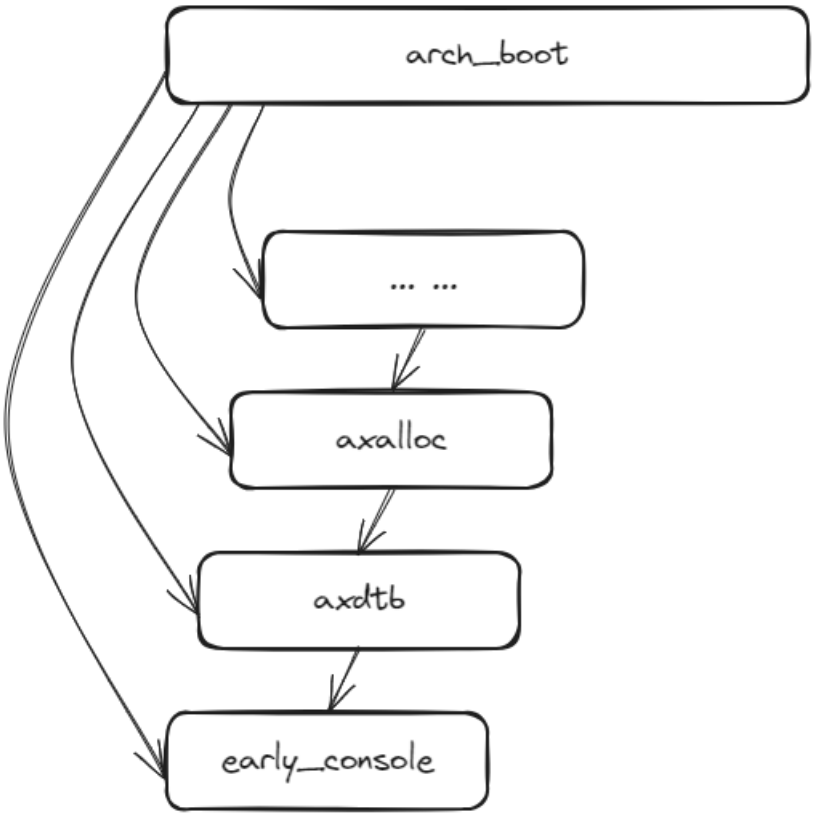
现在的工作策略是：先给出空的syscall实现，涉及特定参数以支持功能时，才针对处理。所以目前对上述syscall的支持仅是支持当前ltp mmap测试用例的要求，并不完善。参考实现来自：LinuxKernel、Starry和蚂蚁的框内核Asterinas。

下步重点是把内部sys_clone的功能暴露出来，以支持ltp对system(cmd)的调用需要。

正在做的工作

从archboot开始，把组件自底向上独立出来。每个有自己的init方法，可以直接单独构造和测试。

archboot的作用是创建一个最小的组件runtime。它现在有点冗余，需要分拆一下，只保留最小启动部分。另外需要把arceos那个make及脚本稍微精简，用于构造组件级测试。



预期达到的效果：从arch_boot开始，自底向上每次加一个组件，该组件会自动把依赖组件一起关联进来。相当于每次构建都是一个可运行的内核，从小到大，直至构成宏内核、unikenel等。