os_buddy_alloc项目

- **作者**: 杨向原(ID:320220942311) 李宏诚(ID:320220941681)
- **学院**: 信息科学与工程学院
- **专业**: 计算机科学与技术

代码布局

build/构建目录 init/：初始化代码 drv/：设备驱动 excp/：异常处理 mm/：内存管理 proc/：进程管理 fs/：文件系统 CMakeLists.txt 构建规则

CMAKE_C_CFLAGS
本测试在真实运行中没有位置无关代码,没有优化,没有插入调试信息 set(CMAKE_C_FLAGS "-Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -march=loongarch64 -mabi=lp64 -ffreestanding -fno-common -nostdlib -Iinclude -fno-stack-protector -fno-pie -no-pie")

CMAKE_EXE_LINKER_FLAGS
页面大小4096字节,代码加载地址0x9000000000200000

目标系统
sudo apt-get install qemu qemu-user qemu-user-static

构建QEMU硬件模拟器模拟LoongArch架构

代码运行

1. 切换至工作文件夹operating_system_work

```
yxy@yxy-virtual-machine:~$ cd operating_system_work/
yxy@yxy-virtual-machine:~/operating_system_work$ cd kernel/
yxy@yxy-virtual-machine:~/operating_system_work/kernel$
```

2. 编译

原有文件架构如下:

```
yxy@yxy-virtual-machine:~/operating_system_work/kernel$ tree
.
├── drv
│   ├── console.c
│   ├── disk.c
│   └── font.c
├── excp
│   ├── exception.c
│   └── exception_handler.S
├── fs
│   └── xtfs.c
├── include
│   └── xtos.h
├── init
│   ├── head.S
│   └── main.c
├── Makefile
├── mm
│   └── memory.c
└── proc
    ├── ipc.c
    ├── proc0
    │   ├── get_proc0_code.sh
    │   └── proc0.S
    ├── process.c
    └── swtch.S

8 directories, 16 files
```

进行make编译后文件结构如下：

```
yxy@yxy-virtual-machine:~/operating_system_work/kernel$ make all
yxy@yxy-virtual-machine:~/operating_system_work/kernel$ tree
.
├── drv
│   ├── console.c
│   ├── console.d
│   ├── console.o
│   ├── disk.c
│   ├── disk.d
│   ├── disk.o
│   ├── font.c
│   ├── font.d
│   └── font.o
├── excp
│   ├── exception.c
│   ├── exception.d
│   ├── exception_handler.d
│   ├── exception_handler.o
│   ├── exception_handler.S
│   └── exception.o
├── fs
│   ├── xtfs.c
│   ├── xtfs.d
│   └── xtfs.o
├── include
│   └── xtos.h
├── init
│   ├── head.d
│   ├── head.o
│   ├── head.S
│   ├── main.c
│   ├── main.d
│   └── main.o
├── kernel
├── Makefile
├── mm
│   ├── memory.c
│   ├── memory.d
│   └── memory.o
└── proc
    ├── ipc.c
    ├── ipc.d
    ├── ipc.o
    ├── proc0
    │   ├── get_proc0_code.sh
    │   └── proc0.S
    ├── process.c
    ├── process.d
```

```
    ├── process.o
    ├── swtch.d
    ├── swtch.o
    └── swtch.S

8 directories, 41 files
```

可以看到，成功生成了.o中间文件，编译成功

添加的buddy_alloc代码如下所示

```
#define MAX_ORDER 12          // 最大块的阶数
#define N_PAGE 32768          // 128MB的物理内存，共有32768个物理页
#define PAGE_SIZE 4096        // 每页大小为4KB
#define MEM_SIZE (N_PAGE * PAGE_SIZE)  // 总内存大小

// mem_map数组的定义
char mem_map[N_PAGE];  // 每个物理页的状态，0为空闲，1为已占用

// 伙伴系统的结构定义
typedef struct free_block {
    unsigned long order;         // 块的阶数，表示块的大小是2^order
    struct free_block *next;     // 下一个空闲块
} free_block_t;

// 用一个数组记录每个阶数的空闲链表
free_block_t *free_list[MAX_ORDER + 1];  // free_list[0]表示1页块的空闲链表，free_list[1]表示2页块的空闲

extern struct process *current;
extern struct shmem shmem_table[NR_SHMEM];

// 初始化伙伴系统
void buddy_system_init() {
    for (int i = 0; i <= MAX_ORDER; i++) {
        free_list[i] = NULL;  // 初始化所有阶数的空闲链表为空
    }

    // 初始时，整个内存作为一个大的空闲块
    free_block_t *block = (free_block_t *)get_page();
    block->order = MAX_ORDER; // 整个内存块大小为2^12页
    block->next = NULL;

    // 将这个大块添加到最大阶数的空闲链表
    free_list[MAX_ORDER] = block;
}

// 分配内存
void *buddy_alloc(unsigned long order) {
    if (order > MAX_ORDER) {
        panic("Request order exceeds MAX_ORDER.");
    }

    // 从空闲链表中查找合适大小的块
    for (int i = order; i <= MAX_ORDER; i++) {
        if (free_list[i] != NULL) {
            // 找到一个足够大的空闲块
```

```
            free_block_t *block = free_list[i];
            free_list[i] = block->next; // 从链表中移除

            // 如果该块大于请求大小，则拆分
            while (i > order) {
                // 拆分成两个更小的块
                free_block_t *buddy = (free_block_t *)((char *)block + (1 << (i - 1)) * PAGE_SIZE)
                buddy->order = i - 1;
                buddy->next = free_list[i - 1];
                free_list[i - 1] = buddy;

                i--;
            }

            // 返回分配的块的地址
            return (void *)block;
        }
    }

    panic("Out of memory in buddy allocator.");
    return NULL;
}

// 释放内存
void buddy_free(void *ptr, unsigned long order) {
    free_block_t *block = (free_block_t *)ptr;
    block->order = order;

    // 获取块的起始地址
    unsigned long block_start = (unsigned long)block & ~(PAGE_SIZE * (1 << order) - 1);

    // 查找块的伙伴
    unsigned long buddy_start = block_start ^ (PAGE_SIZE * (1 << order));
    free_block_t *buddy = (free_block_t *)buddy_start;

    // 合并伙伴
    if ((mem_map[buddy_start >> 12] == 0) && (buddy->order == order)) {
        // 伙伴为空闲且阶数相同，合并
        free_block_t **list = &free_list[order];
        while (*list != NULL && *list != buddy) {
            list = &(*list)->next;
        }

        // 移除伙伴
        if (*list == buddy) {
            *list = buddy->next;
        }
```

```
        // 合并后插入到当前阶数的空闲链表
        if (block_start < buddy_start) {
            block->order = order + 1;
            block->next = free_list[order + 1];
            free_list[order + 1] = block;
        } else {
            buddy->order = order + 1;
            buddy->next = free_list[order + 1];
            free_list[order + 1] = buddy;
        }
    } else {
        // 伙伴不可合并，直接将块加入空闲链表
        block->next = free_list[order];
        free_list[order] = block;
    }
}

// 获取一页内存（使用伙伴系统）
unsigned long get_page() {
    void *page = buddy_alloc(0);  // 请求分配一个页（阶数为0的块）
    return (unsigned long)page;
}

// 释放一页内存（使用伙伴系统）
void free_page(unsigned long page) {
    buddy_free((void *)page, 0);  // 释放一个页（阶数为0的块）
}

// 初始化内存管理
void mem_init() {
    int i;

    // 初始化mem_map数组
    for (i = 0; i < N_PAGE; i++) {
        if (i >= KERNEL_START_PAGE && i < KERNEL_END_PAGE) {
            mem_map[i] = 1;  // 标记内核区为已占用
        } else {
            mem_map[i] = 0;  // 标记用户区为空闲
        }
    }

    // 初始化伙伴系统
    buddy_system_init();

    // 设置控制寄存器
    write_csr_64(CSR_DMW0_PLV0 | DMW_MASK, CSR_DMW0);
    write_csr_64(0, CSR_DMW3);
    write_csr_64((PWCL_EWIDTH << 30) | (PWCL_PDWIDTH << 15) | (PWCL_PDBASE << 10) | (PWCL_PTWIDTH
```

```
    invalidate();

    // 初始化共享内存
    shmem_init();
}

// 获取页表项
unsigned long *get_pte(struct process *p, unsigned long u_vaddr) {
    unsigned long pd, pt;
    unsigned long *pde, *pte;

    pd = p->page_directory;
    pde = (unsigned long *)(pd + ((u_vaddr >> 21) & 0x1ff) * ENTRY_SIZE);
    if (*pde)
        pt = *pde | DMW_MASK;
    else {
        pt = get_page();
        *pde = pt & ~DMW_MASK;
    }
    pte = (unsigned long *)(pt + ((u_vaddr >> 12) & 0x1ff) * ENTRY_SIZE);
    return pte;
}

// 设置页表项
void put_page(struct process *p, unsigned long u_vaddr, unsigned long k_vaddr, unsigned long attr)
    unsigned long *pte;

    pte = get_pte(p, u_vaddr);
    if (*pte)
        panic("panic: try to remap!");
    *pte = (k_vaddr & ~DMW_MASK) | attr;
    invalidate();
}

// 释放进程的页表
void free_page_table(struct process *p) {
    unsigned long pd, pt;
    unsigned long *pde, *pte;
    unsigned long page;

    pd = p->page_directory;
    pde = (unsigned long *)pd;
    for (int i = 0; i < ENTRYS; i++, pde++) {
        if (*pde == 0)
            continue;
        pt = *pde | DMW_MASK;
        pte = (unsigned long *)pt;
        for (int j = 0; j < ENTRYS; j++, pte++) {
```

```
            if (*pte == 0)
                continue;
            page = (~0xfffUL & *pte) | DMW_MASK;
            free_page(page);
            *pte = 0;
        }
        free_page(*pde | DMW_MASK);
        *pde = 0;
    }
}

// 复制页表
void copy_page_table(struct process *from, struct process *to) {
    unsigned long from_pd, to_pd, from_pt, to_pt;
    unsigned long *from_pde, *to_pde, *from_pte, *to_pte;
    unsigned long page;
    int i, j;

    from_pd = from->page_directory;
    from_pde = (unsigned long *)from_pd;
    to_pd = to->page_directory;
    to_pde = (unsigned long *)to_pd;
    for (i = 0; i < ENTRYS; i++, from_pde++, to_pde++) {
        if (*from_pde == 0)
            continue;
        from_pt = *from_pde | DMW_MASK;
        from_pte = (unsigned long *)from_pt;
        to_pt = get_page();
        to_pte = (unsigned long *)to_pt;
        *to_pde = to_pt | DMW_MASK;
        for (j = 0; j < ENTRYS; j++, from_pte++, to_pte++) {
            if (*from_pte == 0)
                continue;
            page = (~0xfffUL & *from_pte) | DMW_MASK;
            *to_pte = page;
        }
    }
}
```

loongarrch64-toolchain.cmake:

为特定的目标架构（LoongArch64）生成正确的编译和链接指令

.cmake指令解析

1. `set(CMAKE_SYSTEM_NAME Linux)`

- **作用**：指定目标系统的名称。在交叉编译中， `CMAKE_SYSTEM_NAME` 用于告诉 CMake 您正在为哪个操作系统进行编译。
- **解释**：这里设置为 `Linux` ，表示目标系统是 Linux 操作系统。

2. `set(CMAKE_SYSTEM_PROCESSOR loongarch64)`
   - **作用**：指定目标处理器的架构。
   - **解释**：设置为 `loongarch64` ，表示目标架构是 LoongArch64。这对于选择合适的编译器选项和汇编指令至关重要。
3. `set(TOOLCHAIN_PREFIX "../cross-tool/bin/loongarch64-unknown-linux-gnu-")`
   - **作用**：定义交叉编译工具链的前缀路径。
   - **解释**：
     - **相对路径**：这里使用的是相对路径 ../cross-tool/bin/loongarch64-unknown-linux-gnu-。这意味着工具链目录位于当前工具链文件所在目录的上一级目录下的 cross-tool/bin/ 目录中。
     - **前缀**：loongarch64-unknown-linux-gnu- 是工具链命名的前缀，常见于交叉编译工具链。例如，loongarch64-unknown-linux-gnu-gcc 是 C 编译器，loongarch64-unknown-linux-gnu-ld 是链接器。
4. `set(CMAKE_C_COMPILER "${TOOLCHAIN_PREFIX}gcc")``
   - **作用**：指定 C 编译器。
   - **解释**：将 C 编译器设置为交叉编译工具链中的 gcc，即 ../cross-tool/bin/loongarch64-unknown-linux-gnu-gcc。
5. `set(CMAKE_ASM_COMPILER "${TOOLCHAIN_PREFIX}gcc")`
   - **作用**：指定汇编编译器（ASM Compiler）。
   - **解释**：将汇编编译器也设置为交叉编译工具链中的 gcc，即 ../cross-tool/bin/loongarch64-unknown-linux-gnu-gcc。在很多交叉编译环境中，gcc 可以同时处理 C 和汇编文件（如 .S 文件）。
6. `set(CMAKE_LINKER "${TOOLCHAIN_PREFIX}ld")`
   - **作用**：指定链接器。
   - **解释**：将链接器设置为交叉编译工具链中的 ld，即 ../cross-tool/bin/loongarch64-unknown-linux-gnu-ld。
7. `set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)`
   - **作用**：配置 CMake 如何在 CMAKE_FIND_ROOT_PATH 中查找可执行程序（如工具链中的编译器、链接器等）。
   - **解释**：
     - **NEVER**：CMake 在查找可执行程序时，不会在 CMAKE_FIND_ROOT_PATH 指定的路径中查找，只会使用系统默认路径或指定的路径。
8. `set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)`
   - **作用**：配置 CMake 如何在 CMAKE_FIND_ROOT_PATH 中查找库文件。
   - **解释**：
     - **ONLY**：CMake 只会在 CMAKE_FIND_ROOT_PATH 指定的路径中查找库文件，不会在系统默认路径中查找。这确保了链接时使用的是目标系统的库，而不是宿主系统的库。
9. `set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)`
   - **作用**：配置 CMake 如何在 CMAKE_FIND_ROOT_PATH 中查找头文件。
   - **解释**：
     - **ONLY**：CMake 只会在 CMAKE_FIND_ROOT_PATH 指定的路径中查找头文件，不会在系统默认路径中查找。这确保了编译时使用的是目标系统的头文件，而不是宿主系统的头文件。

.cmake文件作用

1.**目标系统和处理器架构**：

`CMAKE_SYSTEM_NAME` 设置为 `Linux` ，表示目标系统是 `Linux` 。
`CMAKE_SYSTEM_PROCESSOR` 设置为 `loongarch64` ，表示目标架构是 `LoongArch64` 。

2.**工具链前缀和编译器设置**：

`TOOLCHAIN_PREFIX` 定义了交叉编译工具链的路径前缀。
`CMAKE_C_COMPILER` 和 `CMAKE_ASM_COMPILER` 分别设置为交叉编译工具链中的 `gcc` 。
`CMAKE_LINKER` 设置为交叉编译工具链中的 `ld` 。

3.**库和头文件查找路径配置**：

禁用程序查找路径中的自动查找，以避免混淆。
指定库和头文件仅从目标系统路径中查找，确保使用的是目标系统的依赖项。
交叉编译器路径配置
检查路径:使用export命令临时更改路径

实践过程问题与解决办法

1.**使用变量绝对路径**

- 相对路径可能会导致构建时路径解析问题，特别是在不同的构建目录结构中。建议使用绝对路径或通过 CMake 变量动态构建路径。

```
set(TOOLCHAIN_PREFIX "/home/bruce/project-c/homework/os/os_buddy_alloc/operating_system/cross-tool
```

2.**验证工具链路径是否正确**：

- 确保工具链中的 gcc 和 ld 可执行文件存在并具有执行权限。

```
ls -l /home/bruce/project-c/homework/os/os_buddy_alloc/operating_system/cross-tool/bin/loongarch64
ls -l /home/bruce/project-c/homework/os/os_buddy_alloc/operating_system/cross-tool/bin/loongarch64
```

3.**环境变量配置正确**

- 如果工具链路径复杂，可以考虑在环境变量中设置工具链前缀，然后在工具链文件中引用这些变量。

```
export LOONGARCH64_TOOLCHAIN_PREFIX=/absolute/path/to/cross-tool/bin/loongarch64-unknown-linux-gnu-
```

```
# loongarch64-toolchain.cmake
set(CMAKE_C_COMPILER "${LOONGARCH64_TOOLCHAIN_PREFIX}gcc")
set(CMAKE_ASM_COMPILER "${LOONGARCH64_TOOLCHAIN_PREFIX}gcc")
set(CMAKE_LINKER "${LOONGARCH64_TOOLCHAIN_PREFIX}ld")
```

4.**CMake 工具链文件的正确使用**

- 确保在运行 CMake 时正确指定工具链文件。

```
mkdir -p build
cd build
cmake -DCMAKE_TOOLCHAIN_FILE=../loongarch64-toolchain.cmake ..
make
```

5.**调试工具链配置**

- 如果构建过程中出现问题，可以启用 CMake 的详细输出以调试工具链配置。

```
# 在 loongarch64-toolchain.cmake 中添加
set(CMAKE_VERBOSE_MAKEFILE ON)
```

- 或者在命令行中启用详细输出：

```
cmake -DCMAKE_TOOLCHAIN_FILE=../loongarch64-toolchain.cmake -DCMAKE_VERBOSE_MAKEFILE=ON ..
make VERBOSE=1
```

链接脚本:linker.ld

控制最终可执行文件或内核映像的内存布局和各个段（Sections）的组织方式

`ENTRY` 指令指定程序入口点

`SECTIONS` 块:定义输出文件的各个段及其在内存中的位置和属性。

- .text 段
  - **内容**：包含所有输入文件中的 .text 和 .rodata 段，即代码和只读数据。
  - **目的**：将所有代码和只读数据放置在 .text 段中，并映射到名为 TEXT 的内存区域。
- .data 段
  - **内容**：包含所有输入文件中的 .data 段，即初始化的数据。
  - **目的**：将所有初始化数据放置在 .data 段中，并映射到名为 DATA 的内存区域。
- .bss 段
  - **内容**：包含所有输入文件中的 .bss 段，即未初始化的数据。
  - **目的**：将所有未初始化的数据放置在 .bss 段中，并映射到名为 DATA 的内存区域。

- /DISCARD/ 段
  - **作用**：丢弃所有输入文件中的 .note 和 .comment 段。这些段通常包含调试信息或编译器注释，对最终的内核映像没有实际用途。
  - **用途**：减小最终可执行文件或内核映像的大小，移除不必要的段。

CMakeLists.txt文件

添加cmake工具用于开发可以更好的处理编译,链接问题

1.**基本设置**

`cmake_minimum_required` 和 `project` 定义了 `CMake` 的最低版本要求和项目名称。
`LANGUAGES C ASM` 指定项目使用 C 和汇编语言。

2.**编译器和链接器选项**

`CMAKE_C_FLAGS` 和 `CMAKE_ASM_FLAGS` 设置了与 Makefile 中 `CFLAGS` 相对应的编译选项。
`CMAKE_EXE_LINKER_FLAGS` 设置了链接选项，包括 `-z max-page-size=4096` 和链接脚本 `-T linker.ld` 。

3.**源文件**

`set(SOURCES ...)` 列出了所有的源文件，包括 .c 和汇编文件 .S。

4.**可执行文件**

`add_executable(kernel ${SOURCES})` 定义了一个名为 `kernel` 的可执行文件，包含所有源文件。
`set_target_properties` 和 `target_link_options` 用于指定链接器脚本和链接选项。
禁用标准库链接，以匹配 Makefile 中的 `-nostdlib` 选项。

5.**包含目录**

`target_include_directories(kernel PRIVATE include)` 指定了头文件所在的目录。

6.**编译选项**

`target_compile_options(kernel PRIVATE -Wall -Wextra)` 添加了额外的编译警告选项。

7.**输出目录**

`set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)` 设置了可执行文件的输出目录为 `build/bin` 。

编译结果:

```
os_buddy_alloc/operating_system/kernel/build$cmake ..
```

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/xxxxxx/os_buddy_alloc/operating_system/kernel/build
```

```
os_buddy_alloc/operating_system/kernel/build$make all
```

```
-- The CXX compiler identification is GNU 11.4.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/xxxxxx/os/os_buddy_alloc/operating_system/kernel/build
```

make之后:

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/bruce/project-c/homework/os/os_buddy_alloc/operating_sy
[ 10%] Building C object CMakeFiles/kernel.dir/drv/console.c.o
[ 20%] Building C object CMakeFiles/kernel.dir/drv/disk.c.o
[ 30%] Building C object CMakeFiles/kernel.dir/drv/font.c.o
[ 40%] Building C object CMakeFiles/kernel.dir/excp/exception.c.o
[ 50%] Building C object CMakeFiles/kernel.dir/fs/xtfs.c.o
[ 60%] Building C object CMakeFiles/kernel.dir/init/main.c.o
[ 70%] Building C object CMakeFiles/kernel.dir/mm/memory.c.o
[ 80%] Building C object CMakeFiles/kernel.dir/proc/ipc.c.o
[ 90%] Building C object CMakeFiles/kernel.dir/proc/process.c.o
[100%] Linking C executable kernel
```