提交评测 2024-05-08 20:48:43 | 评测冷却: 108s

提交

更新

Lab 4-1 Extra

准备工作：创建并切换到 `lab4-1-extra` 分支

请在**自动初始化分支后**，在开发机上依次执行以下命令：

```
$ cd ~/学号
$ git fetch
$ git checkout lab4-1-extra
```

初始化的 `lab4-1-extra` 分支基于课下完成的 `lab4` 分支，并且在 `tests` 目录下添加了 `lab4_msg` 样例测试目录。

题目背景

在 Lab 4 的课下实验中，我们实现了阻塞式的 IPC 机制。接收方在调用接收消息函数 `sys_ipc_recv` 后将进入阻塞态，直到发送方发送消息。而发送方在消息发送失败（接收方未处于接收态）时，也将进入阻塞态，直到接收方处于接收态，消息成功发送。

在 Lab 4-1 Extra 中，我们将基于消息队列实现非阻塞式的 IPC 机制。消息队列是 Linux 系统中实现进程间通信的机制之一，它使用链表维护消息控制块。

题目描述

本题在下发的 `include/msg.h` 中，定义了消息控制块 `struct Msg` 及其状态，以及消息队列的结构体 `Msg_list`：

```
enum {
    MSG_FREE, // 空闲消息控制块
    MSG_SENT, // 消息已发送但尚未被接收
    MSG_RECV, // 消息已被接收
};

struct Msg {
    TAILQ_ENTRY(Msg) msg_link; // 类似 env_link，用于构建消息队列
    u_int msg_tier;            // 消息控制块被使用的次数，用于计算消息标识符
    u_int msg_status;          // 消息控制块状态
    u_int msg_value;           // 发送方传递的具体数值
```

```
};

TAILQ_HEAD(Msg_list, Msg);
```

提交

更新

在下发的 `kern/msg.c` 中，定义了存放消息控制块的 `msgs` 数组，并给出了**生成消息标识符**的函数 `msg2id`。调用 `msg2id` 函数时，传入指向消息控制块的指针，函数将根据消息控制块被使用的次数 `msg_tier` 以及它在 `msgs` 数组中的下标，返回消息对应的标识符。对于同一个消息控制块，生成的消息标识符数值只会增加。

```
inline u_int msg2id(struct Msg *m) {
    return (m->msg_tier) * NMSG + (m - msgs);
}
```

对于消息标识符 `msgid`，可以使用定义的宏函数 `MSGX` 得到消息控制块在 `msgs` 数组中的下标。也就是说，你可以使用 `msgs[MSGX(msgid)]` 获得标识符 `msgid` 对应的消息控制块。

在 `include/env.h` 中，你需要在进程控制块中补充以下定义：

```
struct Env {
    // 省略进程控制块中的已有定义
    struct Msg_list env_msg_list; // 进程收到但尚未处理的消息控制块
    u_int env_msg_value;          // 用于向用户态返回 msg_value
    u_int env_msg_from;           // 用于向用户态返回 msg_from
    u_int env_msg_perm;           // 用于向用户态返回 msg_perm
};
```

在本题中，空闲消息控制块构成空闲链表 `msg_free_list`。对于每个进程，其收到但尚未处理的消息控制块构成链表 `env_msg_list`。本题需要的用户态函数具体实现将在题目要求一节中给出。你只需要在 `kern/syscall_all.c` 中实现以下系统调用：

- 消息发送函数 `int sys_msg_send(u_int envid, u_int value, u_int srcva, u_int perm)`
- 消息接收函数 `int sys_msg_recv(u_int dstva)`
- 消息状态查询函数 `int sys_msg_status(u_int msgid)`

消息发送（`int sys_msg_send(u_int envid, u_int value, u_int srcva, u_int perm)`）

本函数的功能为：

- 根据进程标识符 `envid` 找到目标进程，失败的返回 `-E_BAD_ENV` 。
- 从空闲消息链表头部取出消息控制块，将消息控制块被使用的次数 `msg_tier` 增加 1，更新消息控制块的状态为 `MSG_SENT` ，当无空闲消息控制块时返回 `-E_NO_MSG` 。
- 将相应数据填入消息控制块，并将传递的物理页**引用次数增加** 1。将消息控制块插入目标进程的消息链表尾部，返回消息标识符。

注意：

- 将传递的物理页引用次数增加 1，是为了保证物理页在发送方进程退出后不被回收，此操作是必要的。
- 消息控制块的 `msg_perm` 需要在参数 `perm` 的基础上增加权限位 `PTE_V` 。

消息接收（`int sys_msg_recv(u_int dstva)`）

本函数的功能为：

- 当 `dstva` 不为 `0` ，且不为合法虚拟地址（低于 `UTEMP` 或不低于 `UTOP` ）时返回 `-E_INVAL` 。
- 当进程消息链表为空时，没有需要处理的消息控制块，返回 `-E_NO_MSG` 。
- 从消息链表头部取出消息控制块，并完成以下操作：
  - 当需要传递物理页且 `dstva` 不为 `0` 时，根据消息控制块中的数据，传递物理页面的映射关系。
  - 当需要传递物理页时，将传递的物理页**引用次数减少** 1。
  - 此外需要将消息控制块中传递的数据复制到进程控制块中，用于向用户态传递数据。
- 更新消息控制块的状态为 `MSG_RECV` ，并将其插入空闲消息链表尾部，返回 `0` 表示系统调用正常完成。

注意：

- 与 Lab 4 课下实验中实现的 IPC 机制不同，此系统调用**不可以阻塞当前进程**。

消息状态查询（`int sys_msg_status(u_int msgid)`）

本函数的功能为：

- 根据消息标识符 `msgid` 找到消息控制块。
- 当找到的消息控制块使用 `msg2id` 生成的标识符与 `msgid` 相同时，返回消息控制块记录的状态。

提交评测 2024-05-08 20:48:43 | 评测冷却: 108s

- 以上条件均不满足的说明标识符 `msgid` 尚未被分配，返回 `-E_INVAL` 。

题目要求

提交

更新

1. 在 `include/error.h` 中加入宏 `#define E_NO_MSG 14` 。
2. 在 `include/env.h` 文件开头添加 `#include <msg.h>` ，并在进程结构体 `struct Env` 末尾添加以下代码：

```
struct Msg_list env_msg_list;
u_int env_msg_value;
u_int env_msg_from;
u_int env_msg_perm;
```

3. 在 `kern/env.c` 中的 `env_alloc` 函数中，使用 `TAILQ_INIT` 宏函数初始化消息链表 `env_msg_list` 。
4. 在 `include/syscall.h` 中添加系统调用号 `SYS_msg_send` 、 `SYS_msg_recv` 、 `SYS_msg_status` 。新增的系统调用号应当位于 `MAX_SYSNO` 之前。
5. 在 `user/include/lib.h` 末尾添加以下代码：

```
int syscall_msg_send(u_int envid, u_int value, const void *srcva, u_int perm);
int syscall_msg_recv(void *dstva);
int syscall_msg_status(u_int msgid);

int msg_send(u_int whom, u_int val, const void *srcva, u_int perm);
int msg_recv(u_int *whom, u_int *value, void *dstva, u_int *perm);
int msg_status(u_int msgid);
```

6. 在 `user/lib/syscall_lib.c` 末尾添加以下代码：

```
int syscall_msg_send(u_int envid, u_int value, const void *srcva, u_int perm) {
    return msyscall(SYS_msg_send, envid, value, srcva, perm);
}

int syscall_msg_recv(void *dstva) {
    return msyscall(SYS_msg_recv, dstva);
}

int syscall_msg_status(u_int msgid) {
```

7. 在 `user/lib/ipc.c` 末尾添加以下代码：

提交

更新

```
int msg_send(u_int whom, u_int val, const void *srcva, u_int perm) {
    return syscall_msg_send(whom, val, srcva, perm);
}

int msg_recv(u_int *whom, u_int *val, void *dstva, u_int *perm) {
    int r = syscall_msg_recv(dstva);
    if (r != 0) {
        return r;
    }
    if (whom) {
        *whom = env->env_msg_from;
    }
    if (perm) {
        *perm = env->env_msg_perm;
    }
    if (val) {
        *val = env->env_msg_value;
    }
    return 0;
}

int msg_status(u_int msgid) {
    return syscall_msg_status(msgid);
}
```

8. 在 `kern/syscall_all.c` 中实现题目描述中需要完成的系统调用函数。你可以将以下代码复制到 `syscall_all.c` 中以便于你的实现：

```
int sys_msg_send(u_int envid, u_int value, u_int srcva, u_int perm) {
    struct Env *e;
    struct Page *p;
    struct Msg *m;

    if (srcva != 0 && is_illegal_va(srcva)) {
        return -E_INVAL;
    }
    try(envid2env(envid, &e, 0));
    if (TAILQ_EMPTY(&msg_free_list)) {
        return -E_NO_MSG;
    }

    /* Your Code Here (1/3) */
}
```

```
    if (dstva != 0 && is_illegal_va(dstva)) {
        return -E_INVAL;
    }
    if (TAILQ_EMPTY(&curenv->env_msg_list)) {
        return -E_NO_MSG;
    }

    /* Your Code Here (2/3) */
}

int sys_msg_status(u_int msgid) {
    struct Msg *m;

    /* Your Code Here (3/3) */
}
```

9. 在 `kern/syscall_all.c` 中的 `void *syscall_table[MAX_SYSNO]` 系统调用函数表中为 `SYS_msg_send` 、 `SYS_msg_recv` 、 `SYS_msg_status` 添加对应的内核函数指针。

提示

- 在 `init/init.c` 中的 `mips_init` 函数中，保证会调用 `kern/msg.c` 中的 `msg_init` 函数，将所有消息控制块状态置为 `MSG_FREE` ，并插入空闲消息链表。
- 你可以参照 Lab 4 课下实验中实现的 IPC 机制，完成部分函数逻辑。
- 在实现消息接收函数时，你可能需要使用 `page_decref` 函数。

样例输出 & 本地测试

我们在样例测试中创建了两个进程 `0800` 与 `1001` （16 进制表示的 `env_id` ）。

在样例测试中，进程 `1001` 会首先向进程 `0800` 发送一条仅包含值 2024 的消息，然后等待进程 `0800` 向它发送消息。进程 `0800` 再接收到 `1001` 发来的消息后，会紧接着发送一条包含物理页的消息，消息内容为 `Good luck, have fun!` ，随后等待进程 `1001` 退出。

你可以使用：

- `make test lab=4_msg && make run` 在本地测试样例（调试模式）
- `MOS_PROFILE=release make test lab=4_msg && make run` 在本地测试样例（开启优化）

提交评测 2024-05-08 20:48:43 | 评测冷却: 108s

```
1001: Sending value to 0800...
1001: Message 32 sent successfully
0800: Received value 2024 from 1001
0800: Sending message to 1001...
0800: Message 33 sent successfully
1001: Received message "Good luck, have fun!" from 0800 with permission 80
[00001001] destroying 00001001
[00001001] free env 00001001
i am killed ...
[00000800] destroying 00000800
[00000800] free env 00000800
i am killed ...
```

提交

更新

提交评测 & 评测标准

请在开发机中执行下列命令后，在**课程网站**上提交评测。

```
$ cd ~/学号/
$ git add -A
$ git commit -m "message" # 请将 message 改为有意义的信息
$ git push
```

在线评测时，所有的 `.mk` 文件、所有的 `Makefile` 文件、 `init/init.c` 、 `tests/` 和 `tools/` 目录下的所有文件都可能被替换为标准版本。因此请同学们在本地开发时，**不要**在这些文件中编写实际功能所依赖的代码。

注意： 测试点中可能使用课下实现的 IPC 进行进程间的同步。

具体要求和分数分布如下：

测试点序号	评测说明	分值
1	样例测试	10
2	不测试 `sys_msg_status` ，不传递物理页	15
3	传递物理页，收发进程均不提前退出	20
4	传递物理页，不检查权限	25