



University of Science and Technology of China

loongarch xv6 syscalls implement

An Manual for the loongarch

loongarch xv6

2023 年 6 月 7 日

Chapter 1

系统调用

1.1 系统调用测试

测试视频下载连接：<https://www.aliyundrive.com/s/XbZ2ZcZtp3E>

启动内核：打开项目文件后，可以看到项目中的 **run.sh** 文件，执行下面的命令启动内核。

```
1 cd <项目目录>
2 chmod +x ./run.sh
3 ./run.sh
```

测试方法：运行内核后，运行 **ls** 查看是否有 **run-all.sh** 文件，执行下面的命令将自动化测试所有的系统调用

```
1 sh < run-all.sh
```

下面表格为系统调用测试完成情况

测试名	完成情况	错误原因 (简单记录)
brk	✓	FD 最大值为 100, 而原本设定为 16
chdir	✓	
clone	✓	
close	✓	
dup	✓	
dup2	✓	
execve	✓	
exit	✓	
fork	✓	
fstat	✓	
getdents	✓	
getcwd	✓	
getpid	✓	
getppid	✓	
gettimeofday	✓	
mkdir	✓	
mmap	✓	
mount	x	
munmap	✓	
open	✓	
openat	✓	目录不存在, 由于不是挂载测试, 所以需要手动创建
pipe	✓	
sleep	✓	
times	✓	
read	✓	
umount	x	未完成
uname	✓	
wait	✓	
waitpid	✓	
write	✓	
yield	✓	

表 1.1: 系统调用完成情况

1.2 系统调用添加

为了完成所需要的测试，添加必备的系统调用如下

1.2.1 SYS_getcwd 17

- 函数声明: `syscall(SYS_getcwd, buf, size);`
- 功能: 获取当前工作目录;
- Input: `char *buf`: 一块缓存区, 用于保存当前工作目录的字符串。当 `buf` 设为 `NULL`, 由系统来分配缓存区。; `size`: `buf` 缓存区的大小。
- Return: 成功执行, 则返回当前工作目录的字符串的指针。失败, 则返回 `NULL`。

实现思路

由于文件系统的组成方式为顺序组成, 并且没有从 `file` 到 `path` 的函数, 故手动添加一个通过 `inode` 寻找 `path` 的函数

添加思路: 由于顺序存储, 故考虑使用栈作深度优先遍历, 从根目录开始, 若发现目录则压栈, 并且记录目录名称, 通过判断传入的 `inode` 编号来判断是否查找到对应的文件

```
1 int
2 namepath(struct inode* ip, char* buf, int size)
3 {
4     struct inode *dp, *temp_dp;
5     struct inode *list[10];
6     struct dirent *name_list[10];
7     char *name = buf+size-1;
8     *name = '\0';
9     name = buf;
10    int top = 1, count = 0;
11    memset(list, 0, sizeof(list));
12    list[0] = iget(ROOTDEV, ROOTINO);
13    if(ip == list[0]){
14        *(name++) = '/';
15        return 0;
16    }
```

```

17  else{
18      while(top != 0){
19          dp = list[top-1];
20          if(count == 0){
21              *(name++)= '/';
22          }else{
23              int len = strlen(name_list[top-1]->name);
24              memmove(name,name_list[top-1]->name,len);
25              name += len;
26              *(name++) = '/';
27          }
28          top--;
29          uint off;
30          struct dirent de;
31          for(off = 0; off < dp->size; off += sizeof(de)){
32              if(readi(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof
33                  (de))
34                  panic("dirlookup read");
35              if(de.inum == 0) // inum = 1 means "." or ".."
36                  continue;
37              if(de.inum == 1 && ip->inum != 1){
38                  continue;
39              }
40              name_list[top] = &de;
41              temp_dp = iget(dp->dev,de.inum);
42              if(temp_dp->type == T_DIR)
43                  list[top++] = iget(dp->dev, de.inum);
44              if(de.inum == ip->inum){
45                  int len = strlen(de.name);
46                  memmove(name,de.name,len);
47                  name += len;
48                  return 0;
49              }
50          }
51      }
52      return 0;
53 }

```

1.2.2 SYS_dup3 24

- 功能：复制文件描述符，并指定了新的文件描述符；
- Input: old_fd: 被复制的文件描述符, new_fd: 新的文件描述符
- Return: 成功执行，返回新的 fd。失败，返回-1。

实现思路

1. 获取 old_fd 对应的文件，若失败返回-1
2. 尝试读取指定的 new_fd 及其对应文件
3. 如果读取的 new_fd 不符合要求，比如进程所能持有的文件数量已满，返回-1
4. 如果 new_fd 已经有文件（返回 0），则将已有文件关闭，并把 new_fd 指向新指定的文件，将文件引用次数增加
5. 如果 new_fd 没有已有文件（返回-2），直接加入并将文件引用次数增加

核心代码

```
1 uint64 sys_dup3(void) {
2     struct file *old_f, *new_f;
3     int new_fd;
4
5     struct proc * p = myproc();
6     if(argfd(0, 0, &old_f) < 0)
7         return -1;
8
9     switch(argfd(1, &new_fd, &new_f)){
10         case -1: return -1;
11         case 0: fileclose(new_f);
12         default: p->ofile[new_fd] = filedup(old_f); // ofile 进
                程打开文件表
13     }
14
15     return new_fd;
16 }
```

1.2.3 SYS_rename 26

- 函数声明:
- 功能:
- Input:
- Return

1.2.4 SYS_mkdirat 34

- 函数声明: `syscall(SYS_mkdirat);`
- 功能: 创建目录;
- Input:

-
- | | |
|---|--|
| 1 | <code>dirfd</code> : 要创建的目录所在的目录的文件描述符。 |
| 2 | <code>path</code> : 要创建的目录的名称。如果 <code>path</code> 是相对路径, 则它是相对于 <code>dirfd</code> 目录而言的。如果 <code>path</code> 是相对路径, 且 <code>dirfd</code> 的值为 <code>AT_FDCWD</code> , 则它是相对于当前路径而言的。如果 <code>path</code> 是绝对路径, 则 <code>dirfd</code> 被忽略。 |
| 3 | <code>mode</code> : 文件的所有权描述。详见 <code>man 7 inode</code> 。 |
-

- Return: 成功执行, 返回 0。失败, 返回-1。

核心代码

```
1      uint64
2 mkdirat(int dirfd, char* path, int mode)
3 {
4     struct inode *ep;
5
6     if((ep = create2(path, T_DIR, mode, dirfd,0,0)) == 0){
7         return -1;
8     }
9     iunlock(ep);
10    iput(ep);
11    return 0;
12 }
```

```

13
14 uint64
15 sys_mkdirat(void)
16 {
17     char path[MAXPATH];
18     int fd, mode;
19
20     if(argint(0, &fd) < 0 || argstr(1, path, MAXPATH) < 0 ||
        argint(2, &mode) < 0){
21         return -1;
22
23     }
24     begin_op();
25     if(fd < 0 && fd != AT_FDCWD) {
26         printf("[sys_mkdirat] Invalid file descriptor %d\n", fd);
27         end_op();
28         return -1;
29     }
30
31
32     int t = mkdirat(fd, path, mode);
33     end_op();
34     return t;
35 }

```

1.2.5 SYS__unlinkat 35

- 函数声明: `syscall(SYS__unlinkat);`
- 功能: 移除指定文件的链接 (可用于删除文件);
- Input:

-
- 1 `dirfd`: 要删除的链接所在的目录。
 - 2 `path`: 要删除的链接的名字。如果`path`是相对路径, 则它是相对于`dirfd`目录而言的。如果`path`是相对路径, 且`dirfd`的值为`AT_FDCWD`, 则它是相对于当前路径而言的。如果`path`是绝对路径, 则`dirfd`被忽略。
 - 3 `flags`: 可设置为0或`AT_REMOVEDIR`。
-

- Return: 成功执行, 返回 0。失败, 返回-1。

核心代码

```
1
2
3     static int unlink(char* path,int fd){
4 struct inode *ip, *dp;
5     struct dirent de;
6     char name[DIRSIZ];
7     uint off;
8     if(*path == '.' && *(path+1) != '.'){
9         nameiparent(path, name);
10        dp = myproc()->cwd;
11    }
12    else if ((dp = nameiparent(path, name)) == 0) {
13        return -1;
14    }
15    ilock(dp);
16
17    // Cannot unlink "." or "..".
18    if (namecmp(name, ".") == 0 || namecmp(name, "..") == 0)
19        goto bad;
20    iunlockput(dp);
21    if ((ip = dirlookup(dp, name, &off)) == 0){
22        ilock(dp);
23        goto bad;
24    }
25    ilock(dp);
26    ilock(ip);
27    if (ip->nlink < 1)
28        panic("unlink: nlink < 1");
29    if (ip->type == T_DIR && !isdirempty(ip)) {
30        iunlockput(ip);
31        goto bad;
32    }
33    memset(&de, 0, sizeof(de));
34    if (writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de)
35        )
36        panic("unlink: writei");
```

```

36     if (ip->type == T_DIR) {
37         dp->nlink--;
38         iupdate(dp);
39     }
40     iunlockput(dp);
41     ip->nlink--;
42     iupdate(ip);
43     iunlockput(ip);
44     return 0;
45
46 bad:
47     iunlockput(dp);
48     return -1;
49 }
50
51 uint64
52 sys_unlinkat(void)
53 {
54
55     char path[MAXPATH];
56     char h[MAXPATH];
57     int dirfd, mode;
58     struct file *f = NULL;
59     struct inode *dp = NULL, *ip;
60     begin_op();
61     if (argfd(0, &dirfd, &f) < 0) {
62         if (dirfd != AT_FDCWD) { // != AT_FDCWD
63             end_op();
64             return -1;
65         }
66         dp = myproc()->cwd;
67     } else {
68
69         dp = f->ip;
70         if (dp->type == T_DIR) {
71             end_op();
72             return -1;
73         }
74     }

```

```

75
76     int len;
77     if((len = argstr(1, path, MAXPATH)) <= 0 || argint(2, &mode
78         ) < 0){
79         end_op();
80         return -1;
81     }
82     char *s = path + len - 1;
83     while (s >= path && *s == '/') {
84         s--;
85     }
86     for(int i = 0; i < MAXPATH; i++){
87         h[i] = path[i];
88     }
89     if (s >= path && *s == '.' && (s == path || *--s == '/')) {
90         printf("sys_unlinkat", "illegal path %s\n", path);
91         end_op();
92         return -1;
93     }
94     if ((ip = ename( path, dirfd)) == NULL) {
95         printf("sys_unlinkat", "can namei %s\n", path);
96         end_op();
97         return -1;
98     }
99     //int isdir = S_ISDIR(ip->mode);
100     if (ip->type == T_DIR && mode != AT_REMOVEDIR) {
101         iput(ip);
102         printf("sys_unlinkat", "illegal mode 0x%x against 0x%x\
103             n", mode, ip->type);
104         end_op();
105         return -1;
106     } else if (ip->type != T_DIR && mode == AT_REMOVEDIR) {
107         iput(ip);
108         end_op();
109         return -1;
110     }
111     ilock(ip);
112     if (ip->type == T_DIR && isdirempty(ip) != 1) {

```

```

112         iunlockput(ip);
113         printf("sys_unlinkat", "dir isn't empty\n");
114         end_op();
115         return -1;
116     }
117     getcwd(ip,path,128);
118     int ret ;
119     iunlockput(ip);
120     if(h[0] == '.' && h[1] == '/'){
121         ret = unlink(h+2,dirfd);
122     }
123     else{
124         ret = unlink(h,dirfd);
125     }
126     end_op();
127     return ret;
128 }

```

1.2.6 SYS_openat 56

- 函数声明: `syscall(SYS_openat);`
- 功能: 打开或创建一个文件;
- Input:

-
- 1 **fd**: 文件所在目录的文件描述符。
 - 2 **filename**: 要打开或创建的文件名。如为绝对路径, 则忽略**fd**
 。如为相对路径, 且**fd**是`AT_FDCWD`, 则**filename**是相对于
 当前工作目录来说的。如为相对路径, 且**fd**是一个文件描
 述符, 则**filename**是相对于**fd**所指向的目录来说的。
 - 3 **flags**: 必须包含如下访问模式的其中一种: `O_RDONLY`,
 `O_WRONLY`, `O_RDWR`。还可以包含文件创建标志和文件状态
 标志。
 - 4 **mode**: 文件的所有权描述。
-

- Return: 成功执行, 返回新的文件描述符。失败, 返回-1。

核心代码

```

1 static struct inode*
2 create2(char *path, short type, int mode, int dirfd,int major,
   int minor)
3 {
4     struct inode *ip, *dp;
5     char name[DIRSIZ + 1];
6     if((dp = enameparent(path, name, dirfd)) == 0)
7         return 0;
8     ilock(dp);
9     if ((ip = dirlookup(dp, name, 0)) != 0) {
10         iunlockput(dp);
11         ilock(ip);
12         if (type == T_FILE && (ip->type == T_FILE || ip->type
            == T_DEVICE))
13             return ip;
14         iunlockput(ip);
15         return 0;
16     }
17     if ((ip = ialloc(dp->dev, type)) == 0) {
18         iunlock(dp);
19         iput(dp);
20         return 0;
21     }
22     ilock(ip);
23     ip->major = major;
24     ip->minor = minor;
25     ip->nlink = 1;
26     iupdate(ip);
27     if ((type == T_DIR && !(ip->type == T_DIR)) ||
28         (type == T_FILE && (ip->type == T_DIR))) {
29         iunlock(dp);
30         iput(ip);
31         iput(dp);
32         return 0;
33     }
34     if (type == T_DIR) { // Create . and .. entries.
35         dp->nlink++;      // for ".."
36         iupdate(dp);
37         // No ip->nlink++ for ".": avoid cyclic ref count.

```

```

38         if (dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..",
39             dp->inum) < 0)
40             panic("create dots");
41     }
42     if (dirlink(dp, name, ip->inum) < 0)
43         panic("create: dirlink");
44     iunlockput(dp);
45     return ip;
46 }
47 uint64 openat(int dirfd, char* path, int flags, int mode)
48 {
49     struct file *f;
50     struct inode *ep;
51     int fd;
52     if(flags & O_CREATE) {
53         ep = create2(path, T_FILE, flags, dirfd, 0, 0);
54         if(ep == 0) {
55             return -1;
56         }
57     } else {
58         if((ep = ename(path, dirfd)) == 0) {
59             printf("[sys_openat]No such file or directory!! %s\n",
60                 path);
61             return -1;
62         }
63         ilock(ep);
64     }
65     if(flags == O_DIRECTORY){
66         ep->type = T_DIR;
67     }
68     if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
69         if (f) {
70             fileclose(f);
71         }
72         iunlock(ep);
73         iput(ep);
74         return -1;
75     }
76     if(!(ep->type == T_DIR) && (flags & O_TRUNC)){

```

```

75     itrunc(ep);
76 }
77 f->type = FD_INODE;
78 f->off = (flags & 0x004) ? ep->size : 0;
79 f->ip = ep;
80 f->readable = !(flags & O_WRONLY);
81 f->writable = (flags & O_WRONLY) || (flags & O_RDWR);
82
83 iunlock(ep);
84
85 return fd;
86 }
87 uint64
88 sys_openat(void)
89 {
90     char path[MAXPATH];
91     int fd, flags, mode;
92     if(argint(0, &fd) < 0 || argstr(1, path, MAXPATH) < 0 ||
93         argint(2, &flags) < 0 || argint(3, &mode) < 0 )
94         return -1;
95     begin_op();
96     if(fd < 0 && fd != -100) {
97         printf("[sys_openat]Invalid file descriptor %d\n", fd);
98         end_op();
99         return -1;
100     }
101     int ret = openat( fd, path, flags, mode);
102     end_op();
103     return ret;
104 }

```

1.2.7 SYS_sched_yield 124

- 函数声明: syscall(SYS_sched_yield);
- 功能: 让出调度器;
- Input: 系统调用 ID;
- Return: 系统调用 ID;

实现思路

直接使用 xv6 的 yield 函数即可

1.2.8 SYS_times 153

- 函数声明: syscall(SYS_times, tms);
- 功能: 获取进程时间;
- Input:tms 结构体指针, 用于获取保存当前进程的运行时间数据;
- Return: 成功返回已经过去的滴答数, 失败返回-1;

实现思路

在 proc 结构体中添加 user_time,kernel_time, 分别在用户态时钟中断和内核态时钟中断时记数, 添加 cutime,cstime, 在每次发生时钟终端时, 将进程的子进程的运行时间叠加, 故定义 proc_tick 函数

```
1 void proc_tick(void) {
2     struct proc* p;
3     struct proc* np;
4     for(p = proc; p < &proc[NPROC]; p++) {
5         acquire(&p->lock);
6         p->user_time += 1;
7         p->kernel_time += 1;
8         np = p->parent;
9         while(np){
10             np->cstime += p->kernel_time + p->cstime;
11             np->cutime += p->user_time + p->cutime;
12             np = np->parent;
13         }
14         release(&p->lock);
15     }
16 }
```

1.2.9 SYS_uname 160

- 函数声明: syscall(SYS_uname, buf)

- 功能：将系统信息放入用户态
- Input: utsname 结构体指针
- Return: 成功返回 0，失败返回-1;

实现思路

添加结构体定义，使用 memmove 函数将内容复制到结构体的地址中，最后将信息从 kernel space 复制到 user space

1.2.10 SYS__gettimeofday 169

- 函数声明: syscall(SYS__gettimeofday, ts, 0);
- 功能：获取时间；
- Input: timespec 结构体指针用于获得时间值；
- Return: 成功返回 0，失败返回-1;

实现思路

增加 r_time 汇编代码，使用 rdtsc 来得到龙心指令系统中的时钟

1.2.11 SYS__getppid 173

- 函数声明: syscall(SYS__getppid)
- 功能：获取父进程 pid
- Input: void
- Return: 父进程 pid

实现思路

通过查找进程是否具有父进程，来返回对应的 pid

1.2.12 SYS_brk 214

- 功能：修改进程堆大小，在实际内核中，用于回收内存空间
- Input: 作为新地址的内存地址
- Return 0 表示修改成功，-1 表示修改失败

实现思路

通过新的 addr，调用 growproc 即可

1.2.13 SYS_munmap 215

- 函数声明：syscall(SYS_munmap, char* addr , int len);
- 功能：取消一个文件在内存中的映射；
- Input: addr 是映射的地址，当 addr 为 0 时，由操作系统来决定最终映射的地址（本实验中 addr 总是 0）；length 是映射的内存长度；
- Return 成功则直接返回 0；失败则返回-1；

实现思路

```
1 //sysfile.c
2 //...
3 //int munmap( char* addr , int len );
4 uint64 sys_munmap(void)
5 {
6     // to gain the arguments and proc
7     uint64 addr;
8     int len;
9     if( argaddr( 0 , &addr ) < 0 || argint( 1 , &len ) < 0 )
10         return -1;
11     struct proc* p = myproc();
12     //to find the target vma
13     struct VMA* vp = 0;
14     for( int i=0 ; i<VMA_MAX ; i++ )
15         if( p->vma[i].addr <= addr && addr < p->vma[i].addr + p->
            vma[i].len && p->vma[i].valid == 1 )
```

```

16     {
17         vp = &p->vma[i];
18         break;
19     }
20     if( vp == 0 )
21         panic("munmap no such vma");
22     // if the page has been mapped
23     if( walkaddr( p->pagetable , addr ) != 0)
24     {
25         //write back if necessary
26         if( vp->flags == MAP_SHARED )
27             filewriteoff( vp->f , addr , len , addr-vp->addr ); //
                this function is new
28         uvmunmap( p->pagetable , addr , len/PGSIZE , 1 );    //
                unmap
29         return 0;
30     }
31     // maintain the ref of file and the valid of the vma
32     if( 0 == (vp->mapcnt -= len) )
33     {
34         fileclose( vp->f );
35         vp->valid = 0;
36     }
37     return 0;
38 }

```

其中 write back 用到的 filewriteoff 函数是一个对 fwrite 进行模仿写的一个函数，xv6 原生的 fwrite 只能从 idx=0 处开始写

```

1  //file.c
2  //...
3  int filewriteoff(struct file *f, uint64 addr, int n , int off)
4  {
5      int r, ret = 0;
6      if(f->writable == 0)
7          return -1;
8      if(f->type == FD_INODE){
9          int max = ((MAXOPBLOCKS-1-1-2) / 2) * BSIZE;
10         int i = 0;
11         while(i < n){

```

```

12     int n1 = n - i;
13     if(n1 > max)
14         n1 = max;
15     begin_op();
16     ilock(f->ip);
17     if ((r = writei(f->ip, 1, addr + i, off, n1)) > 0)
18         off += r;
19     iunlock(f->ip);
20     end_op();
21     if(r != n1){
22         // error from writei
23         break;
24     }
25     i += r;
26 }
27 ret = (i == n ? n : -1);
28 } else {
29     panic("my filewrite");
30 }
31 return ret;
32 }

```

fork&exit

在 fork 以及 exit 中对当前进程的 vma 字段进行复制、删除，以通过 fork 情况下的测试。

```

1  //proc.c
2  //...
3  int fork(void)
4  {
5  //...
6
7      np->maxaddr = p->maxaddr;
8      for( int i=0 ; i<VMA_MAX ; i++ )
9          if( p->vma[i].valid )
10             {
11                 filedup( p->vma[i].f );
12                 memmove( &np->vma[i] , &p->vma[i] , sizeof( struct VMA )
13                             );
14             }
15 }

```

```

14     return pid;
15 }
16 //...
17 void exit(int status)
18 {
19     struct proc *p = myproc();
20
21     for( int i=0 ; i<VMA_MAX ; i++ )
22     {
23         if( p->vma[i].valid == 1 )
24         {
25             struct VMA* vp = &p->vma[i];
26             for( uint64 addr = vp->addr ; addr < vp->addr + vp->len
                ; addr += PGSIZE )
27             {
28                 if( walkaddr( p->pagetable , addr ) != 0 )
29                 {
30                     if( vp->flags == MAP_SHARED )
31                         filewriteoff( vp->f , addr , PGSIZE , addr-vp->
                            addr );
32                     uvmunmap( p->pagetable , addr , 1 , 1 );
33                 }
34             }
35             fileclose( p->vma[i].f );
36             p->vma[i].valid = 0;
37         }
38     }
39 //...

```

1.2.14 SYS_clone 220

- 函数声明: `syscall(SYS_clone, fn, stack, flags, NULL, NULL, NULL);`
- 功能: 创建一个新的进程
- Input: flags: 创建的标志, 如 SIGCHLD; stack: 指定新进程的栈, 可为 0;
- Return: 成功返回 clone 的进程 pid, 失败返回-1

实现思路

该系统调用的实现同 fork, 所以在 fork 函数的基础之上, 添加对栈的支持。需要在 proc.c 中添加 clone 函数, 大体实现方法同 fork

1.2.15 SYS__execve 221

- 函数声明: `syscall(SYS__execve, name, argv, argp);`
- 功能: `execve` 为内核级系统调用, 执行程序
- Input: 参数 `name` 字符串所代表的文件路径, 第二个参数是利用数组指针来传递给执行文件, 并且需要以空指针 (NULL) 结束, 最后一个参数则为传递给执行文件的新环境变量数组。
- Return: 函数执行成功时没有返回值, 执行失败时的返回值为-1.

实现思路

该系统调用的实现同 `exec`, 不同的是需要将参数从用户空间复制到系统空间, 然后在 `exec.c` 文件中添加 `execve` 函数, 具体实现和 `exec` 相差不大, 将三个参数传递的环境变量数组处理后放在 `tramfram` 的 `a2` 处作为输入即可

1.2.16 SYS__mmap 222

- 函数声明: `char* mmap(char* addr, int len, int prot, int flags,int fd, int off);`
- 功能: 实现其 `memory-mapped files` 功能, 也就是把一个文件来映射到内存中;
- Input: 其中 `addr` 是映射的地址, 当 `addr` 为 0 时, 由操作系统来决定最终映射的地址 (本实验中 `addr` 总是 0); `length` 是映射的内存长度; `prot` 是映射对应的权限 (`read/write`); `flags` 是映射的类型, 分为 `shared` 和 `private` , 如果是 `shared` , 那么最终对文件进行的修改会写回外存; `fd` 是文件描述符; `off` 是偏移量 (本实验中 `off` 总是 0)。
- Return 返回映射到的内存空间的地址

实现思路

```
1 //proc.h
2 #define VMA_MAX 16
3 struct VMA{
4     int valid;           //有效位，当值为 0 时表示无效，即为 empty
5     element
6     uint64 addr;        //记录起始地址
7     int len;            //长度
8     int prot;           //权限 (read/write)
9     int flags;          //区域类型 (shared/private)
10    int off;             //偏移量
11    struct file* f;      //映射的文件
12    uint64 mapcnt;       //(延迟申请) 已经映射的页数量
13 };
14 struct proc {
15     //...
16     struct VMA vma[VMA_MAX];    // virtual memory address field
17     arr
18 //...
19 };
```

对应的，我们需要在初始化进程的时候初始化进程的 vma 中的一些信息。

```
1 //proc.c
2 static struct proc*
3 allocproc(void)
4 {
5     //...
6     found:
7     //...
8     //to init the vma field
9     for( int i=0 ; i<VMA_MAX ; i++ )
10    {
11        p->vma[i].valid = 0;
12        p->vma[i].mapcnt = 0;
13    }
14    p->maxaddr = MAXVA - 2*PGSIZE;
15    return p;
16 }
```

从 trapframe 的底部向下生长，我们需要在 struct proc 中维护一个新的变量来记录 heap 区的可用部分最大地址。

```
1 //proc.h
2 struct proc {
3 //...
4     struct VMA vma[VMA_MAX];      // virtual memory address field
        arr
5     uint64 maxaddr;                // virtual memory to be assigned
        to the vma
6 };
```

mmap 的实现

```
1 //sysfile.c
2 //...
3 //char* mmap( char* addr, int length, int prot, int flags,int
        fd, int off);
4 uint64 sys_mmap(void)
5 {
6     // to gain the arguments , file and proc
7     uint64 addr;
8     int len , prot , flags , fd , off;
9     if( argaddr( 0 , &addr ) < 0 || argint( 1 , &len ) < 0 ||
        argint( 2 , &prot ) < 0 || argint( 3 , &flags ) < 0 ||
        argint( 4 , &fd ) < 0 || argint( 5 , &off ) < 0 )
10         return -1;
11     struct proc* p = myproc();
12     struct file* f = p->ofile[fd];
13
14     // to ensure the prot
15     if( ( flags == MAP_SHARED && f->writable == 0 && (prot&
        PROT_WRITE)) )
16         return -1;
17
18     // to find a empty vma and init it
19     int idx;
20     for( idx = 0 ; idx < VMA_MAX ; idx++ )
21         if( p->vma[idx].valid == 0 )
22             break;
23     if( idx == VMA_MAX )
```



```

24     panic("no empty vma field");
25
26     struct VMA* vp = &p->vma[idx];
27     vp->valid = 1;
28     vp->len = len;
29     vp->flags = flags;
30     vp->off = off;
31     vp->prot = prot;
32     vp->f = f;
33     filedup( f ); //increase the ref of the file
34     vp->addr = (p->maxaddr-=len); // assign a useable virtual
        address to the vma field , and maintain the maxaddr
35     return vp->addr;
36 }

```

缺页中断的处理，通过查阅 loongarch 相关手册实现。

```

1  else if((r_csr_estat() >> 16) == PIL || (r_csr_estat() >> 16)
    == PIS) {
2      // printf("=== trap ===\n");
3      uint64 addr = r_csr_badv();
4      struct VMA* vp = 0;
5      //to find the target vma
6      for( int i=0 ; i<VMA_MAX ; i++ ) {
7          if( p->vma[i].addr <= addr && addr < p->vma[i].addr + p
            ->vma[i].len && p->vma[i].valid == 1 )
8              {
9                  vp = &p->vma[i];
10                 break;
11             }
12     }
13     if( vp != 0 )
14     {
15         uint64 mem = (uint64)kalloc();
16         memset( (void*)mem , 0 , PGSIZE );
17         if( -1 == mappages( p->pagetable , addr , PGSIZE , mem
            , PTE_U | PTE_V | ( vp->prot << 1 ) ) )
18             panic("pagefault map error");
19
20         vp->mapcnt += PGSIZE; //maintain the mapcnt

```

```

21         ilock( vp->f->ip );
22         readi( vp->f->ip , 1 , addr , addr-vp->addr , PGSIZE);
           //copy a page of the file from the disk
23         iunlock( vp->f->ip );
24     }
25 }

```

1.2.17 SYS_wait4 260

- 函数声明: `syscall(SYS_wait4, pid, status, options);`
- 功能: 等待进程改变状态;
- Input: `pid`: 指定进程 ID, 可为-1 等待任何子进程; `status`: 接收状态的指针; `options`: 选项: `WNOHANG`, `WUNTRACED`, `WCONTINUED`;
- Return 成功则返回进程 ID; 如果指定了 `WNOHANG`, 且进程还未改变状态, 直接返回 0; 失败则返回-1;

实现思路

1. 如果 `pid` 不合法, 返回-1
2. 如果 `pid` 不属于子进程, 返回-1
3. 若子进程已经结束, 将状态返回
4. 若没结束, 若 `options` 为 `WNOHANG` 直接返回, 否则进行放弃 `cpu`。

核心代码

```

1 int
2 waitpid(int ch_pid, uint64 addr, int options)
3 {
4     if (ch_pid == 0 || ch_pid < -1) return -1;
5     // if (ch_pid == -1) return wait(addr);
6
7     struct proc *np;
8     int flag;
9     struct proc *p = myproc();

```

```

10  int ret = -1;
11
12  // hold p->lock for the whole time to avoid lost
13  // wakeups from a child's exit().
14  acquire(&p->lock);
15
16  for(;;){
17      // Scan through table looking for exited children.
18      flag = 0;
19      for(np = proc; np < &proc[NPROC]; np++){
20          // this code uses np->parent without holding np->lock.
21          // acquiring the lock first would cause a deadlock,
22          // since np might be an ancestor, and we already hold p->
           lock.
23          if (np->parent != p) continue;
24
25          if (ch_pid > 0 && np->pid != ch_pid) continue;
26
27          // if(np->parent == p && np->pid == ch_pid){
28              // np->parent can't change between the check and the
               acquire()
29              // because only the parent changes it, and we're the
               parent.
30          acquire(&np->lock);
31          if(np->state == ZOMBIE){
32              // Found one.
33              flag = np->pid;
34              ret = np->xstate << 8;
35              if(addr != 0 && copyout2(addr, (char *)&ret, sizeof(ret
                )) < 0) {
36                  release(&np->lock);
37                  release(&p->lock);
38                  return -1;
39              }
40              freeproc(np);
41              release(&np->lock);
42              release(&p->lock);
43              return flag;
44          } else {

```

```

45         flag = 1;
46     }
47     release(&np->lock);
48     // }
49 }
50
51 // No point waiting if we don't have any children.
52 if(!flag || p->killed){
53     release(&p->lock);
54     return -1;
55 }
56
57 if (options & WNOHANG) return 0;
58
59 // Wait for a child to exit.
60 sleep(p, &p->lock); //DOC: wait-sleep
61 }
62 }
63
64 sys_waitpid(void)
65 {
66     int pid, options;
67     uint64 status;
68     if(argint(0, &pid) < 0){
69         return -1;
70     }
71     if(argaddr(1, &status) < 0){
72         return -1;
73     }
74     if(argint(2, &options) < 0){
75         return -1;
76     }
77     return waitpid(pid, status, options);
78 }

```
