

# 1. 对核心文件的改动分析

---

## defs.h

### 作用：

是定义各种常量、结构体、宏和函数声明的地方。在实现信号量和优先级调度时，相关的数据结构和系统调用的原型声明会在此文件中定义。

### 改动：

增加了信号量相关的函数声明 (`initsem`, `sys_sem_create`, `sys_sem_p`, `sys_sem_v`, `sys_sem_free` 等)。

增加了与进程优先级调度相关的函数声明，如 `chpri` 和 `wakeuplp`。

定义了一个全局变量 `sh_var_for_sem_demo`，是用来测试信号量的机制。

## main.c

### 作用：

作为系统启动文件，其中初始化了一些必要的系统资源，并启动了内核的主调度循环。在信号量和优先级调度相关的功能中，被修改以设置进程调度策略，并且在系统初始化时注册新的系统调用。

### 改动：

在 `main()` 函数中增加了 `initsem()` 的调用，这在操作系统启动时会初始化信号量机制。

在 `startothers()` 和 `mpmain()` 中，CPU 启动的流程没有变化，但它们依赖于新的信号量和调度机制。

## Makefile

### 作用：

负责项目的编译和链接过程。在修改了核心文件后，需要更新以确保新的文件和改动能够被正确编译和链接。保证信号量和优先级调度的代码能够被正确编译，并将其集成到最终的系统中。

### 改动：

将 `UPROGS` 列表扩展了，添加了几个新的用户程序（如 `_testprio`, `_testshrw`, `_testshrwplus`），这些是用来测试优先级调度和信号量功能的测试程序。

## proc.c / proc.h

### 作用：

在这两个文件中定义了进程管理的实现，包括进程的创建、调度、切换等。我们修改了这些文件来实现进程优先级调度，并修改了进程结构体以支持优先级字段。

## 改动：

### 1. allocproc 函数修改

我们在 found 位置添加了 `p->priority = 10;`，为每个新进程设置了一个默认优先级。

```
found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->priority = 10; // 设置默认优先级为10
    release(&ptable.lock);
```

这一修改为每个新进程提供了一个默认优先级。后续调度时，可以根据这个优先级来决定进程的执行顺序。

### 2. scheduler 函数修改

我们根据进程优先级进行调度，并且引入了 `prio` 的循环来遍历不同优先级的进程。修改后的 `for` 循环部分如下：

```
for (prio = 0; prio < 20; prio++) { // 优先级从0到19
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state != RUNNABLE) // 跳过不可运行状态的进程
            continue;

        if (p->priority != prio) // 跳过优先级不匹配的进程
            continue;

        // 选择进程运行
        c->proc = p;
        switchvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        switchkvm();
        c->proc = 0;
    }
}
```

这个修改允许调度器优先选择具有较高优先级（较低 `prio` 值）的进程。我们使用了双重循环，第一层循环遍历优先级（`prio`），第二层循环遍历进程表，选择优先级匹配且可运行的进程。以下是几点需要注意的地方：

- 我们的优先级调度是一个简化的策略，优先级低的进程会一直等待优先级高的进程先运行。
- 当前的优先级仅适用于 `RUNNABLE` 状态的进程，没有考虑到其他状态的进程（如 `SLEEPING`）。后续可以进一步完善调度策略，确保 `SLEEPING` 状态的进程也能被适当处理。

### 3. procdump 函数修改

在 `procdump` 函数中，我们做了以下修改将进程的优先级输出：

```
cprintf("\nPID=%d state=%s prio=%d %s :", p->pid, state, p->priority, p->name);
```

这使得能够在 `procdump` 输出中看到每个进程的优先级。这个修改有助于调试和观察系统的进程状态。

## 4. chpri 函数

这个函数允许你根据进程ID调整进程的优先级：

```
int chpri(int pid, int priority) {
    struct proc* p;
    acquire(&ptable.lock); // 获取进程表锁

    // 查找指定PID的进程
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) { // 匹配PID
            p->priority = priority; // 修改优先级
            break;
        }
    }

    release(&ptable.lock); // 释放进程表锁
    return pid; // 返回PID
}
```

这个函数允许动态修改进程的优先级。不过需要输入验证，确保优先级范围在合理的范围内（比如0到19）。如果进程表中没有找到对应的进程，函数会返回 -1 并且做额外的处理。

## 5. wakeup1p 函数

wakeup1p 函数在进程处于 SLEEPING 状态时，检查其是否符合 chan 条件，并将其状态设置为 RUNNABLE，唤醒它：

```
void wakeup1p(void *chan) {
    acquire(&ptable.lock); // 获取进程表锁
    struct proc *p;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state == SLEEPING && p->chan == chan) {
            p->state = RUNNABLE;
            break; // 唤醒第一个符合条件的进程
        }
    }
    release(&ptable.lock); // 释放进程表锁
}
```

### 进程优先级字段的添加（proc 结构体中）

我们在 proc.h 中的 proc 结构体中添加了 int priority 字段，这为进程调度引入了优先级概念。通过为每个进程赋予优先级，可以在调度时根据优先级来选择下一个执行的进程，这将有助于更高效地管理多任务环境。

通过这些改动，系统能够实现优先级调度，优先级高的进程会先于优先级低的进程被调度执行。

## spinlock.h / spinlock.c

### 作用：

这两个文件是用于实现内核中同步的基本工具，在多进程操作和资源访问时防止并发问题。在信号量机制的实现时，需要改动这两个文件来保护共享资源并且保证信号量在并发环境下的正确性，防止数据竞争和访问冲突。

### 改动：

在 `spinlock.c` 中，我们修改了锁的结构以及相关函数，使其支持信号量的操作。我们添加了一个新的结构体 `sem`，用来表示信号量，并对 `spinlock` 做了调整，主要体现在：

- **信号量资源计数 ( `resource_count` )**：每个信号量管理一个资源计数，当资源被获取时，计数减少，当资源被释放时，计数增加。
- **`procs` 字段**：这是一个存储进程的数组，用于管理哪些进程在等待信号量。
- **信号量的创建、销毁和操作**：通过 `sys_sem_create`、`sys_sem_free`、`sys_sem_p` 和 `sys_sem_v` 系统调用，我们能够创建信号量，处理进程对信号量的请求（例如 P 操作和 V 操作）。和管理

#### 信号量的实现细节：

实现的信号量操作如下：

- **信号量创建 ( `sys_sem_create` )**：为信号量分配一个编号，并初始化相关的资源计数。
- **信号量释放 ( `sys_sem_free` )**：释放信号量，确保该信号量不再被使用。
- **P 操作 ( `sys_sem_p` )**：用于请求信号量，如果资源不足（计数为负），进程会被阻塞，直到资源可用。
- **V 操作 ( `sys_sem_v` )**：用于释放信号量，释放资源并唤醒等待该信号量的进程。

在 `spinlock.h` 中，我们修改了结构体定义，引入了 `SEM_MAX_NUM`，并添加了一个全局变量 `sh_var_for_sem_demo`，这有助于管理最大信号量数量。这样做的好处是可以定义系统中可用的信号量数量，并通过修改该常量来调节系统的资源管理。

## sysproc.c

### 作用：

负责系统调用中与进程管理相关的实现。在这里，我们新增了与进程优先级调度相关的系统调用逻辑。通过对 `sysproc.c` 的修改，系统能够根据进程的优先级调度进程执行，增强了系统对进程优先级的支持。

### 改动：

在 `sysproc.c` 文件的末尾，新增了几个处理新系统调用的函数：

#### `sys_chpri`

```
int sys_chpri(void)
{
    int pid, pr;
```

```

    if (argint(0, &pid) < 0) {
        return -1;
    }

    if (argint(1, &pr) < 0) {
        return -1;
    }

    return chpri(pid, pr);
}

```

- **功能：**`sys_chpri` 是用于通过系统调用调整进程的优先级。该函数从用户空间获取目标进程的 PID 和目标优先级（`pid` 和 `pr`），并调用内核中的 `cpri` 函数进行优先级调整。
- **参数：**
  - `pid`：要调整优先级的目标进程的 PID。
  - `pr`：目标优先级（通常，较小的数字表示较高的优先级）。
- **作用：**实现了一个简单的优先级调度接口，允许用户程序动态修改进程的优先级。

### `sys_sh_var_read`

```

int sys_sh_var_read()
{
    return sh_var_for_sem_demo;
}

```

- **功能：**`sys_sh_var_read` 直接读取共享变量 `sh_var_for_sem_demo` 的值。
- **作用：**这是一个简单的读取共享资源的函数，可能用于测试进程间的同步机制或信号量的功能。

### `sys_sh_var_write`

```

int sys_sh_var_write()
{
    int n;
    if (argint(0, &n) < 0) {
        return -1;
    }

    sh_var_for_sem_demo = n;

    return sh_var_for_sem_demo;
}

```

- **功能：**`sys_sh_var_write` 将用户传入的值 `n` 写入共享变量 `sh_var_for_sem_demo`。
- **作用：**此函数为共享资源的写操作提供了接口，允许用户空间程序修改共享变量的值。它可能与信号量相关，用于控制访问共享资源的进程同步。

## 信号量系统调用 (`sys_sem_*`)

- `sys_sem_create`：创建一个新的信号量实例，并返回其标识符。
- `sys_sem_free`：释放一个已创建的信号量。
- `sys_sem_p`：执行 P 操作，即申请信号量并阻塞进程直到信号量可用。
- `sys_sem_v`：执行 V 操作，即释放信号量，唤醒等待的进程。

这些函数可能会被后续的内核代码和进程调度器使用，确保在多进程环境下正确地进行同步。

## user.h / usys.S/syscall.c / syscall.h

### 作用：

`user.h`提供了用户程序使用的头文件，`usys.S`用于封装系统调用的汇编层实现，`syscall.c` 和 `syscall.h` 负责系统调用的实现和声明，修改这两个文件以实现新的系统调用。

### 改动：

#### 用户空间的接口 (`user.h`)

在 `user.h` 中，我们为信号量相关的函数和进程优先级相关的函数添加了声明。通过这些声明，可以在应用程序中直接调用这些系统调用。例如：

- `chpri(int pid, int pr)`：修改进程优先级。
- `sem_create(int n)`：创建信号量，`n` 是初始资源数。
- `sem_p(int id)`：执行 P 操作，等待信号量。
- `sem_v(int id)`：执行 V 操作，释放信号量。
- `sem_free(int id)`：释放信号量。
- `sh_var_read()` 和 `sh_var_write(int n)`：读取和写入共享变量。

#### 系统调用的汇编接口 (`usys.S`)

在 `usys.S` 中，我们为新的系统调用（如 `chpri`、`sem_create` 等）添加了相应的汇编代码。这些代码将系统调用号 (`SYS_chpri`、`SYS_sem_create` 等) 存入 `eax` 寄存器，并通过中断 `int $T_SYSCALL` 来触发系统调用处理程序。

#### 添加了新的系统调用 (`syscall.c`和`syscall.h`)

新增系统调用定义：

- `SYS_chpri 23`：此系统调用编号用于进程优先级调整，调用该系统调用可以改变指定进程的优先级。
- `SYS_sh_var_read 24` 和 `SYS_sh_var_write 25`：这两个系统调用用于读写共享变量 `sh_var_for_sem_demo`，用于演示信号量的使用或调度过程中的共享资源访问。
- `SYS_sem_create 26`、`SYS_sem_free 27`、`SYS_sem_p 28`、`SYS_sem_v 29`：这些是与信号量相关的系统调用，分别用于创建信号量、释放信号量、获取信号量（P操作）和释放信号量（V操作）。

## 2. 完成功能测试

### 三个测试文件和测试结果分析:

#### testprio.c

- **测试内容:** 此文件演示了如何通过设置进程优先级，并创建多个子进程来测试优先级调度。
  - **测试流程:**
    1. 设置当前进程的优先级为 19（较高优先级）。
    2. 创建多个子进程，每个子进程的优先级被设置为 15 或 5（较低优先级）。
    3. 通过 `chpri` 修改不同进程的优先级，打印出每个进程的优先级，观察进程的执行顺序。
    4. 运行后，观察是否高优先级的进程会优先执行，低优先级进程是否会被抢占。
  - **原理:** 该测试验证了系统的优先级调度是否工作正常。根据 `chpri` 的设置，调度器应当优先执行高优先级进程，并且应该在高优先级进程执行完之前，低优先级进程不会获得 CPU 时间。
  - **结果分析:**

```
pid=6 running
ppid=5 running
id=7 started
pppid=7 running
id=8 started
pid=8 running
id=4 started
pid=4 running
pid=5 running
pid=6 running
pid=7 running
pid=8 running
pid=4 running
pid=6 finished 6132
pid=7 finished 7161
pidp=8 id=5 finished 5105
finished 0
pid=4 finished 4080
zombie!
zombie!
zombie!
zombie!
```

#### 进程的创建与运行:

程序输出了一系列进程的创建与执行信息:

```
pid=5 running
pid=7 started
pid=8 started
...
pid=6 running
pid=4 running
...
```

这表明多个进程被依次创建并进入了运行状态（running）。

#### 每个进程的运行完成和结果:

运行结果显示了一些进程完成执行的状态以及计算结果:

```
pid=4 finished 6132
pid=7 finished 7161
pid=8 finished 5105
...
```

这里可以看到每个进程完成后打印了其 pid 和计算结果，表明测试过程中每个进程都完成了其任务，并输出了独立的结果值。

程序在运行结束时打印了若干 "zombie?"：

```
zombie?
zombie?
...
```

这说明在进程执行完成后，出现了一些未被父进程回收的僵尸进程。

僵尸进程的出现可能是测试逻辑的一个检查点，旨在验证调度器和进程管理器的行为。

需要确保父进程能够及时回收子进程，避免资源泄漏。

各个进程的计算结果不同，表明它们的计算逻辑是独立的。

进程间没有发生资源冲突或干扰。

## testshrw.c

- **测试内容：**此文件与 testprio.c 很相似，但它更多地关注共享资源的同步。
  - **测试流程：**
    1. 类似于 testprio.c，首先设置进程的优先级。
    2. 创建多个子进程，并通过 chpri 调整它们的优先级。
    3. 测试信号量机制，确保在进程间访问共享资源时能够同步，避免并发问题。
    4. 进程通过 sem\_p 和 sem\_v 获取和释放信号量，以确保共享资源的安全访问。
  - **原理：**此测试验证了优先级调度和信号量的协同工作。信号量保证了进程间的同步，优先级调度保证了高优先级进程的优先执行，二者共同作用确保系统在多进程环境下能够正确运行。
  - **结果分析**

已上传的图片

父进程和子进程在运行时，分别执行自己的逻辑，并通过信号量进行同步，避免同时访问共享资源。

每次访问共享变量时，进程会使用 sem\_p 获取信号量，确保当前访问独占；访问完成后，通过 sem\_v 释放信号量，允许其他进程继续访问。

sum = 118357 和 sum = 126371 分别是父进程和子进程分别运行其逻辑后的结果。

由于信号量的互斥机制，确保了每个进程访问资源时不会发生数据竞争，因此每个 sum 的结果是正确的，且独立的。

从结果来看，两个进程的输出没有乱序，说明信号量和优先级调度协同工作良好。



# testshrwplus.c

- **测试内容：**该测试主要展示了父子进程如何使用信号量进行同步。
  - **测试流程：**
    1. 创建一个初始值为 1 的信号量。
    2. 父进程和子进程交替获取和释放信号量，确保每个进程按顺序打印出信息。
    3. 每个进程在执行时使用 `sem_p` 来获取信号量，执行完任务后使用 `sem_v` 来释放信号量。
    4. 进程通过 `sleep` 来模拟任务的执行，并确保任务的执行顺序不受竞争影响。
  - **原理：**此测试验证了信号量的正确性。信号量保证了父子进程在访问共享资源时的互斥性。父进程和子进程通过交替执行来避免竞争条件。
  - **结果分析**

```
170844 bytes (171 kB, 167 KiB) copied, 0.00232207 s, 73.6 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=
raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 50
init: starting sh
$ testshrwplus
create 0 sem
[Parent] Iteration 0
[Child] Iteration 0
[Parent] Iteration 1
[Child] Iteration 1
[Parent] Iteration 2
[Child] Iteration 2
[Parent] Iteration 3
[Child] Iteration 3
[Parent] Iteration 4
[Child] Iteration 4
free 0 sem
test finished

buddy total pages: dc00
buddy queues (4k-4M):0 0 0 0 0 0 0 0 37
Total RAM size: 224MB
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32
t 50
init: starting sh
$ testshrwplus
create 0 sem
[Parent] Iteration 0
[Child] Iteration 0
[Parent] Iteration 1
[Child] Iteration 1
[Parent] Iteration 2
[Child] Iteration 2
[Parent] Iteration 3
[Child] Iteration 3
[Parent] Iteration 4
[Child] Iteration 4
free 0 sem
Test finished.
$ @77777
```

信号量的创建：

图片中的

```
create 0 sem
```

这表示成功创建了一个信号量，ID 为 0，其初始值为 1（默认为互斥锁信号量）。

父子进程的交替执行：

```
[Parent] Iteration 0
[Child] Iteration 0
[Parent] Iteration 1
[Child] Iteration 1
[Parent] Iteration 2
[Child] Iteration 2
[Parent] Iteration 3
[Child] Iteration 3
[Parent] Iteration 4
[Child] Iteration 4
```

从输出来看，父进程和子进程在执行任务时严格交替打印 `[Parent] Iteration x` 和 `[Child] Iteration x`，表明父进程和子进程在访问共享资源时没有发生冲突。这是信号量 `sem_p` 和 `sem_v` 操作确保的结果。

### 信号量释放：

```
free 0 sem
```

在父子进程任务完成后，信号量被释放，表示信号量的生命周期结束，成功清理了信号量资源。

### 测试结束：

```
Test finished.
```

测试完成，没有报错或其他异常。