深 圳 大 学 实 验 报 告

课程名称： 操作系统

实验项目名称：综合实验一 xv6入门实验+调度+信号量

学院： 计算机与软件学院

专业： 计算机科学与技术

指导教师： 罗秋明

报告人：谭嘉豪 学号： 2020152087 班级： 数计班

实验时间： 2023年4月13日

实验报告提交时间： 2023年5月11日

教务部制

|  |
| --- |
| 一、实验目的与要求   1. 加深对进程调度的直观认识； 2. 掌握xv6操作系统中调度代码的编码实现方法； 3. 掌握xv6操作系统中信号量的编码实现；   二、实验内容   1. 可以使用Linux+Qemu仿真环境； 2. 修改xv6内核代码实现基本系统调用； 3. 修改xv6内核代码实现指定的调度方法； 4. 修改xv6内核代码实现进程间同步所需的信号量机制。   三、实验步骤及说明   1. Helloworld程序 2. 修改xv6操作系统启动时z的提示信息   在kernek/main.c程序的if语句中添加一句printf("xv6 is running!");  在xv6 目录的 qemu-loongarch-runenv 下执行./run\_loongarch.sh -k ../kernel/kernel启动 QEMU 仿真器并运行 xv6 操作系统，在原来的 shell 文本窗口显示 xv6 的输出，可以观察到上述添加的启动信息。   1. 添加Helloworld程序   在xv6的源码的user目录下，编写一个Helloworld.c程序，代码如下：  其中types.h、stat.h和user.h 分别是kernel和user本目录中的头文件。程序运行结果是打印一行信息“Hello, world\n”。  然后修改Makefile文件中的UPROGS变量，添加一个$/\_Helloworld，然后执行make all进行编译。  重新启动xv6系统，执行ls命令可以观察到Helloworld，执行Helloworld程序，输出“Hello, world”。   1. 实现系统调用   xv6系统中可以使用的系统调用均在user/user.h中定义，在程序中可以直接使用，以下为user/user.h中一部分已经实现的系统调用，通过编写程序使用进行验证（以getpid()为例）。  编写如下print-pid.c程序：  编译并运行print-pid，成功打印进程号。  接下来新增系统调用getcpuid()。  第一步：增加系统调用号  由于xv6的系统调用都有一个唯一编号，定义在kernel/syscall.h中。因此可以在SYS\_close的后面，新加入一行“#define SYS\_getcpuid 22”即可，这里的编号22可以是其他值，满足不与其他系统调用的编号冲突即可。  第二步：增加用户态入口  修改user.h，在user/user.h加入函数原型声明“int getcpuid(void);”，  定义了getcpuid()原型之后，还需要实现getcpuid()函数。我们在user/usys.pl中加入一行“entry("getcpuid");”  第三步：修改kernel/syscall.h中的跳转表  声明外部函数sys\_getcpuid()，添加“extern uint64 sys\_getcpuid(void);”。  在系统调用公共入口syscall()中，xv6将根据系统调用号进行分发处理。负责分发处理的函数syscall()（定义于kernel/syscall.c），分发依据是一个跳转表。在跳转表中加入“[SYS\_getcpuid] sys\_getcpuid,”。（下标22对应的是sys\_getcpuid()函数地址）  通过上述操作，已经可以用getcpuid()作为系统调用户态的入口，而且进入系统调用的分发例程syscall()中也能正确地转入到sys\_getcpuid()函数里，接下来只需要在kernel/sysproc.c实现sys\_getcpuid()函数，在kernel/proc.c中实现内核态的getcpuid()函数。  sys\_getcpuid()函数如下：  getcpuid()函数如下：  最后，为了让sysproc.c中的sys\_getcpuid()能调用proc.c中的getcpuid()，还需要在kernel/defs.h加入一行“int getcpuid(void); ”，用作内核态代码调用getcpuid()时的函数原型。  最后，验证新增系统调用是否能被应用程序所正常使用，编写print-cpuid.c程序如下：  编译并运行，正确打印cpuid，成功添加系统调用。  （二）调度实验   1. 调整时间片长度   尝试将一个进程运行的时间片扩展为N个时钟周期，具体思路：每个进程的PCB中添加时钟计数值，当前进程的时间片未用完则不切换。  在xv6的进程控制块kernel/proc.h中修改proc结构体，增加成员slot并定义时间片长度为 8 个tick。  然后在proc.c文件中，找到用于创建进程时分配结构体的allocproc()函数，设置新进程的时间片初值为SLOT。  为了能查看到进程时间片信息，还需要在kernel/proc.c中的 procdump()函数中将输出信息增加一项时间片剩余量。  xv6 原本是在每次时钟中断时就调用 yield()让出 CPU并引发一次调度。现在修改后的代码 需要对时间片剩余量进行递减，以及判定当前进程时间片是否用完——决定是否需要进行调度。  在 trap.c 的 usertrap()函数中完成上述检查，添加的代码如下：  最后查看修改后的时间片信息。  编写loop.c程序，创建两个进程，分别执行长时间的循环计算。代码如下：  编译并运行，在loop运行时，就可以用Ctrl+P检查当前进程剩余的时间片。   1. 优先级调度 2. 增加优先级属性   在kernel/proc.h文件中修改proc结构体，添加成员priority。  在创建进程时设置一个默认优先级，在用于创建进程分配proc结构体的allocproc()函数中设置默认优先级为10。  为了能查看进程的优先级，我们需要修改proc.c中的procdump()函数，使之能打印优先级信息。   1. 添加系统调用   在kernel/syscalll.h中添加新的系统调用编号。  在user/user.h中添加用户态函数原型int chpri(int, int)函数，参数列表为[进程号，新的优先级]。  在user/usys.pl中，添加 chpri()函数的汇编实现代码。  修改kernel/syscall.h中的跳转表，添加[SYS\_chpri] sys\_chpri。  在kernel/syscall.h中定义外部函数extern uint64 sys\_chpri(void);  在 kernel/sysproc.c 中实现 sys\_chpri()，代码如下：  在proc.c文件中实现chpri()函数，代码如下：  最后在defs.h的proc.c部分将添加函数原型“uint64 chpri(int,int);”，以便内核代码访问该函数。   1. 修改调度器   为进程添加优先级的信息后，还需要在调度器中修改调度行为，原调度行为：按顺序遍历进程结构体数组，找到第一个处于RUNNABLE状态的进程并将状态修改为RUNNING。  在proc.c文件中的scheduler()函数进行修改调度行为，修改后的调度行为：首先找到所有进程的优先级中的最高优先级，向后找到与最高优先级相等的就绪进程，或者完成一轮循环后选取目前优先级最高的程序。  在proc.c文件中，将for循环体修改为如下，添加531，532，535-546行代码，选出处于就绪状态且优先级最高（priority最小）的进程进行执行。   1. 验证优先级   创建一个优先级为5的子进程，，然后查看它们被调度的情况。  编写如下程序，使用fork()函数创建子进程，使用chpri()函数修改父进程和子进程的优先级，编译并运行。  使用Ctrl+P查看进程调度的实时状态，可以看到优先级较高的子进程（pid=10）比优先级较低的父进程（pid=9）率先进行执行，子进程变为run状态，父进程为runble状态。  当子进程sleep，父进程又开始执行，父进程状态变为run，子进程状态变为sleep。  当子进程再次运行，又抢占父进程进行执行，子进程变为run状态，父进程为runble状态。  当子进程运行结束后，父进程再次运行，由于父进程没有执行wait()系统调用，子进程状态变为zombie，父进程运行至结束。   1. 信号量 2. 共享变量   在kernel文件夹下新建sem.h文件，添加一个共享变量sh\_var\_for\_sem\_demo。  接下来添加两个系统调用：sh\_var\_read(void)和sh\_var\_write(int)。   1. 访问共享变量   在syscall.h文件中添加系统调用编号。  在user.h文件中添加用户态函数原型。  在 usys.pl 末尾插入两行“entry("sh\_var\_read");”和“entry("sh\_var\_write");”。  修改系统调用跳转表，添加以下两个元素。  在syscall.c的syscalls[]数组前面声明上述两个函数是外部函数。  最后在 sysproc.c 中实现 sh\_var\_read()和 sh\_var\_write()函数。  首先添加为sysproc.c添加头文件sem.h。  添加如下代码：   1. 互斥的并发访问   在定义了共享变量及其访问方法后，尝试编写程序并访问共享变量。  编写sh\_rw\_look.c，代码如下：  运行sh\_rw\_look.c程序，可以发现得到的结果与正确结果（200000）不相等。   1. 信号量数据结构   为了实现信号量，除了创建、撤销、P、V 操作外，还需要添加新的数据结构、初始化函数、调整 wakeup 唤醒操作等。  在kernel/spinlock.h中，声明一个struct sem结构体，且声明信号量总数不超过128个。sem结构体成员包括：   * resource\_count：记录信号量中资源的数量。 * lock：内核自旋锁，为了让信号量的操作保持原子性。 * allocated用于表示该信号量是否已经被分配使用，1表示已分配，0表示未分配。   声明外部变量sem\_used\_count表示正在使用的信号量数目。  声明外部结构体数组sems表示系统拥有的信号量数目。   1. 信号量操作的系统调用   为避免修改Makefile文件，将信号量实现的核心代码放置在kernel/spinlock.c文件中。   * seminit()   由于系统启动时需要对信号量进行初始化操作，编写一个seminit()函数，其工作为：完成信号量数组的自旋锁的初始化。  在 kernel/main.c 的 main()中插入一行“seminit()； //semaphor”（插在 userinit() 之前）。为了让 main.c 能调用 seminit ()，还需要在 defs.h 中插入 seminit ()函数原型。   * sys\_sem\_create()   sys\_sem\_create()扫描 sems[]数组，查看里面 allocated 标志，发现未用的则将其 allocated 置 1，即可返回其编号。如果扫描一次后未发现，则返回错误代码。注意每次操作时需要对 sems[i]进行加锁操作，检查完成后进行解锁操作。   * sys\_sem\_free()   sys\_sem\_free()将指定 id 作为下标访问 sems[id]获得当前信号量 sems[id]，然后对 sems[id].lock 加锁，判定该信号量上没有睡眠阻塞的进程，则将 sems[id].allocated 标志设置为 未使用，从而释放信号量，最后对 sems[id].lock 解锁。   * sys\_sem\_p()   sys\_sem\_p()将指定 id 作为下标访问 sems[id]获得当前信号量 sem，然后用 acquire()对 sems[id].lock 加锁，加锁成功后对sems[id].resource\_count--，接着用realease()解锁退出临界区。如果发现 sems[id].resource\_count< 0则睡眠。其他情况下则直接返回表示完成 p 操作。  注意在 sleep()的时候，会释放 sems[id].lock 才执行 sched()切换——允许其他进程继续执 行 P 操作或 V 操作。而 sleep()返回前，会再次持有sems[id].lock——即使有多个等待进程被唤醒，也只有一个进程能被唤醒并退出睡眠阻塞状态。   * sys\_sem\_v()   sys\_sem\_v()将指定 id 作为下标访问 sems[id]获得当前信号量 sem，然后对 sem.lock 加锁， 加锁成功后对 sem.resource\_count+=1，如果发现 sem.resource\_count>=0，则解锁 sem.lock，并 唤醒该信号量上阻塞的睡眠进程。否则直接返回。  其余添加系统调用的辅助代码与先前添加系统调用的代码步骤一致，这里不再阐述。   1. 修正wakeup操作   由于xv6的wait()系统调用会将所有等待相同事件的进程唤醒，因此需要重写一个新的唤醒函数wakeup1p()，用于唤醒等待指定信号量的一个进程。  在proc.c中实现wakeup1p()函数，且在defs.h在声明函数原型。   1. 用户测试代码   重新编写一个访问共享变量的应用程序sh\_rw\_lock，并且加上信号量的互斥控制。  修改Makefile文件，添加“$U/\_sh\_rw\_lock\”。  sh\_rw\_lock.c程序代码如下：  、  运行程序，可以发现得到了正确的结果200000。   1. 简单的slab机制内核内存管理   在vm.c文件在定义结构体slab，用于存储每个slab的信息，包括每个object的大小、object的数量、slab的起始地址、使用标志（表示某个object是否被使用）。  初始定义8个slab。  在vm.c中编写初始化slab的程序，slab大小为16、32、64、128、256、512、1024、2048共8种尺寸。  接下来实现分配和回收的系统调用。   * 分配函数slab\_allocate()   对于申请的每一块内存大小size（size>=0且size<=2048），为其寻找一个合适的slab块，在该slab块上寻找到第一个没有被使用的object块，将其分配，修改该块的状态变为已被使用，然后计算object块所在物理地址，将虚拟地址所在页映射到计算出的物理地址上，最后返回object块在物理页帧上的偏移量。   * 回收函数slab\_free()   根据虚拟地址，使用walkaddr函数在页表中查找到相应的物理页地址，取虚拟地址的低12位页内偏移与其相加得到物理地址，再查找slab与相应的object位置，更改使用状态，解除页表映射。    在defs.h中加入：  在main.c在调用slabinit()函数。  接下来是添加两个系统调用。  在defs.h中添加：  在syscall.h中添加系统调用号：  在syscall.c的跳转表中添加：  在跳转表上面声明外部函数：  在user.h中添加用户态入口：  编写测试代码slab\_test.c：  申请三块32字节的内存后，释放第二次申请的内存块，再次申请一次。  在xv6启动时，打印slab的所有信息，最初无内存块被使用。  运行程序，得到如下结果：  最后一次申请的内存块物理地址为第二次申请后又释放的内存块的物理地址，成功实现简单的slab内存管理机制。  四、感想及其他  Xv6的代码实现是非常难看懂，每走一步都需要花费我较多的时间去理解代码，并利用网络寻找资料，但也收获良多，通过真正对一个操作系统内核代码进行理解和修改，增加了我对课本上许多理论知识的理解。 |

深圳大学学生实验报告用纸

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  2023年 月 日 |
| 备注： |

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后10日内。