2020 年秋操作系统 xv6 源码阅读报告 2

进程模型

黎善达

1800012961@pku.edu.cn

2020 年 10 月 31 日

1 关键代码阅读与分析

xv6 源码中，涉及进程模型的主要文件包括 proc.h，proc.c，swtch.S；同时涉及一些其它文件中的宏和函数。文件 vm.c 中的函数涉及较多，文件 kalloc.c 中的函数也有一定涉及，但课程中尚未讲授对应内容，因此此处仅在遇到相应函数时对具体的函数进行说明，而不逐个单独分析这两个文件中的全部函数。

proc.h

该文件的主要内容是一些结构的定义。

该文件定义了结构 cpu，用于刻画一个 CPU 的状态信息。这些信息包括局部可编程中断控制器的 ID、含有调度程序入口的上下文、全局描述符表、中断嵌套的深度、当前的中断使能、指向当前正在运行的进程的指针等。这里的许多变量将在 proc.c 的代码中反复出现，例如，上下文切换过程中，struct context *scheduler（即含有调度程序入口的上下文）将发挥重要作用。

该文件定义了结构 context，即上下文。这里定义的上下文包括寄存器%edi、%esi、%edx、%ebp 和%eip。之所以不包含%cs 等段寄存器，是因为段寄存器在内核的上下文中是恒定不变的。之所以不包含%eax、%ecx、%edx，是因为在 x86 规范中，它们是调用者保存寄存器。这里定义的上下文会被按照次序保存在它们所描述的栈的栈底。

该文件定义了枚举类型 procstate，即进程状态。可以看出，**xv6 中的进程共有 6 种状态**，分别是 UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE。

该文件定义了结构 proc，用于刻画一个进程。proc 结构是本次报告中最核心的数据结构。proc 结构包含的信息有：

- 与该进程的内存、打开文件等资源相关的信息：内存大小、页表指针、内核栈底指针、打开文件表。
- 描述该进程的信息：进程号、该进程的父进程（的指针）、进程当前目录。
- 与中断、调度相关的信息：中断帧（的指针）、上下文（的指针）。
- 该进程的状态：是否睡眠、是否已被杀死。
- 因为先前调用了 `yield` 函数，所以需要再次检查当前进程是否被杀死：若当前进程的权限为 `DPL_USER` 且 `killed` 位已被置 1，则退出程序。

该文件的最后通过注释的形式说明了进程内存的内容：进程的内存是连续的，从低地址到高地址分别是 text 段，原始数据和 bss 段，固定大小的栈，可扩展的堆。

proc.c

该文件包含了大量与进程模型相关的基本函数。

该文件声明了进程表 `ptable`，它包含了一个自旋锁，和一个含有 `NPROC` 个 `proc` 结构的数组。宏 `NPROC` 在 `param.h` 中被定义为 64，即支持的最大进程数是 64。该文件中还有函数 `void pinit(void)`，其行为是初始化 `ptable` 中的自旋锁。

该文件定义了一些简单的辅助函数。函数 `struct cpu* mycpu(void)` 通过在中断控制器 ID 的列表中查找当前进程所在 CPU 的中断控制器 ID，确定当前的 CPU。该函数必须在关中断的条件下调用，以免在函数执行过程中遇到重新调度。函数 `int cpuid()` 返回当前进程所在 CPU 的编号。函数 `struct proc* myproc(void)` 返回指向当前进程的指针，其中因为调用 `mycpu()` 而必须实现关中断，调用结束后恢复之。

下面分析该文件中较为主要的函数，将简要总结函数的行为，并指出其中值得注意的细节。

`static struct proc* allocproc(void)`。

该函数会完成创建进程时所需要进行的部分初始化工作。一个进程被真正“创建”前，在进程表中实际上处于 `UNUSED` 状态（因为进程表 `ptable` 将被初始化为全为 0）。该函数从头遍历进程表（此前需要获得进程表的锁），找到的第一个状态为 `UNUSED` 的进程（找到后，释放进程表的锁），分如下步骤进行初始化：

- 将状态置为 `EMBRYO`，为该进程分配进程号。
- 调用 `kalloc()` 函数为该进程分配内核栈空间。`kalloc()` 函数在文件 `kalloc.c` 中定义，其行为是分配一个 4096 比特的物理内存，并返回一个内核可用的指针。
- 移动栈指针，留出一个中断帧的空间。
- 设置新的上下文，令新创建的进程执行开始时从 `forkret()` 函数返回，且返回后进入 `trapret()` 函数（从而用到先前分配的中断帧）。`proc.c` 稍后将定义 `forkret()` 函数；`trapret()` 函数已经在中断异常机制部分解读了。

- 对调用 allocproc() 的函数，返回新进程的指针。

需要指出，allocproc() 的函数调用并不一定总是成功，返回 0 即代表失败。若遍历进程表后没有找到状态为 UNUSED 的进程，将释放进程表的锁并返回 0。若调用 kalloc() 函数为新进程分配内核栈空间失败，则将新进程的状态置回 UNUSED 并返回 0。

void userinit(void)。

该函数将创建第一个用户进程。首先,该函数调用 allocproc() 找出一个进程;其次,调用 kalloc.c 中定义的 setupkvm() 和 inituvm(pde_t *pgdir, char *init, uint sz)) 函数，分内核和用户两部分创建页表；然后为新进程设置上下文环境，将程序计数器设置为 initcode.S 的起始地址；最后，该文件完成新进程的名称、目录的设置，将新进程状态置为 RUNNING。需要注意的是，执行将新进程状态置为 RUNNABLE 的语句前需要将进程表上锁，因为该语句并非一个原子语句，我们不希望该语句执行到一半新进程就因状态改变而被调度上 CPU 了。

int growproc(int n)。

该函数将当前进程的内存增大 n 个比特；n 也可以取负值，代表将当前进程内存减小。两种情况下，该函数将分别调用 kalloc.c 中定义的 allocuvm(pde_t *pgdir, uint oldsz, uint newsz) 和 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz) 修改页表以达到目的，这两个函数都不要求作为参数的新旧内存大小是页对齐的。从具体实现上说，这两个函数分别调用 kalloc 函数和 kfree 函数一页一页地增加或减少内存以使之达到预期大小。

int fork(void)。

该函数会通过复制当前进程以创建一个以当前进程为父进程的新进程，并通过设置进程的栈使得如同从系统调用返回。该函数的流程分为以下几步：

- 调用 kalloc() 函数分配一个新进程。
- 调用 kalloc.c 中定义的 copyuvm 函数，将当前进程的用户内存赋值给新进程。
- 设置新进程的内存空间、父进程。
- 将当前进程的中断帧指针复制给新进程，并将其中对应%eax 寄存器的位置改为 0，新进程最终从 trapret“返回”时，得到的返回值是 0。
- 将父进程的打开文件表、名称复制给新进程；将新进程状态置为 RUNNING。再次注意，执行将新进程状态置为 RUNNABLE 的语句前需要将进程表上锁。
- 父进程得到的返回值为新进程的进程号。

需要指出，fork() 的函数调用并不一定总是成功，返回-1 即代表失败。若遍历进程表后没有找到状态为 UNUSED 的进程，将返回-1。若调用 copyuvm 函数复制用户内存失败，则将新进程的状态置回 UNUSED，释放已分配的内核内存并返回-1。

void exit(void)。

该函数会退出当前进程，且不会返回。但已退出的进程不会直接被清除，而是进入 ZOMBIE 状态

等待其父进程调用 wait() 函数（稍后会详细分析）发现该进程退出。该函数首先进行一个进程退出时所必须执行的工作，例如关闭打开文件等；然后调用 wakeup1() 函数，唤醒处于数遍状态的父进程；最后将自己的子进程的父进程改为初始进程，若此时发现自己的子进程处于 ZOMBIE 状态，则会调用 wakeup1() 函数唤醒初始进程。以上工作完成后，该进程将自身状态置为 ZOMBIE，进入调度程序，不再返回。初始进程不能调用 exit() 函数，否则该函数调用 panic 报告错误并崩溃。

int wait(void)。

该函数会等待当前进程的一个子进程退出，回收这一子进程并返回这一子进程的进程号。若当前进程没有子进程，则该函数返回-1。具体来说，该函数反复遍历进程表寻找 ZOMBIE 状态的子进程，一旦找到，释放该进程的内核栈和页表空间，将记录的各项信息（如名称、内存大小等）置为 0，将其状态置为 UNUSED 并返回。若遍历一次进程表发现当前进程无子进程，或当前进程已经被杀死，则返回-1；若发现有子进程而为进入 ZOMBIE 状态，则调用 sleep 函数进入睡眠状态等待唤醒。由于此处涉及遍历进程表，因此在遍历前需要先获得进程表的锁。

void scheduler(void)。

该函数是各个 CPU 的调度程序。该函数不会返回，而是先进行一些所必须的设置，然后通过 swtch 函数在被调度的函数和自身之间不断交换控制。具体来说，该函数首先会关中断，获得进程表的锁，其次按次序遍历进程表，寻找状态为 RUNNABLE 的进程，将当前进程状态改为 RUNNING，调用 switchuvm 进入用户内存，将当前 CPU 进程改为被选中的进程，用 swtch 函数进行上下文切换（同时相当于将控制权转给被选中的进程）。当被调度的进程让出 CPU 时，控制返回 scheduler 函数，该函数调用 switchkvm 进入内核内存，进行下一轮调度。调用 scheduler 函数的进程需要自行释放进程表锁，并在回到 scheduler 函数前重新获得之，并自行修改进程状态。

void sched(void)。

该函数调用 swtch，从被调度程序回到 scheduler 函数。该函数确保当前进程有进入 scheduler 函数的条件，如已获得进程表锁，进程状态已被修改、中断嵌套恰有一层等；同时，该函数还保存当前 CPU 的中断使能信息，因为中断使能信息实际上是当前进程的性质而非 CPU 的性质。

void yield(void)。

该函数使得当前进程获得进程表锁，将自身状态改为 RUNNABLE，然后调用 sched 函数使系统重新调度进程，达到让出 CPU 的目的。

void forkret(void)。

如同 allocproc 函数所设置的，一个进程的第一次被调度将会进入此处，这里将 scheduler 函数中获得的进程表锁释放，然后返回。返回地址也已经被 allocproc 函数设置好了：即 trapret 函数，这使得当前进程“回到”用户空间。

void sleep(void *chan, struct spinlock *lk)。

该函数修改当前进程状态和结构中的 chan 成员表明该进程进入睡眠状态，然后调用 sched 函数

使系统重新调度进程。

static void wakeup1(void *chan)。

该函数会唤醒所有处在睡眠态，且 chan 成员值为该函数参数的进程。具体而言，该函数遍历进程表，逐个检查进程，将符合条件的进程状态改为 RUNNABLE。调用该函数时，必须已经获得进程锁。

void wakeup(void *chan)。

该函数首先获得进程锁，然后调用 wakeup1 函数，最后释放进程锁。若需要唤醒部分处在睡眠态的进程而未获得进程锁，可以直接调用该函数。

int kill(int pid)。

该函数会杀死以参数为进程号的进程。其具体实现是，遍历进程表，逐个比对进程号，将进程号与参数相等的进程的 killed 位置 1；若被杀掉的进程处于睡眠态，将之改为 RUNNABLE，最后返回 0。若没有找到合乎要求的进程，则返回-1。该函数由于需要遍历进程表，所以首先需要获得进程表锁，并且返回前需要释放之。在中断异常机制部分的代码阅读中，我们看到，被杀掉的进程不会立即退出，而是在回到用户态时才在 trap 函数中执行 exit 函数退出。

void procdump(void)。

该函数供调试使用，将会打印所有状态不为 UNUSED 状态的进程的进程号、状态、名称等信息。

swtch.S

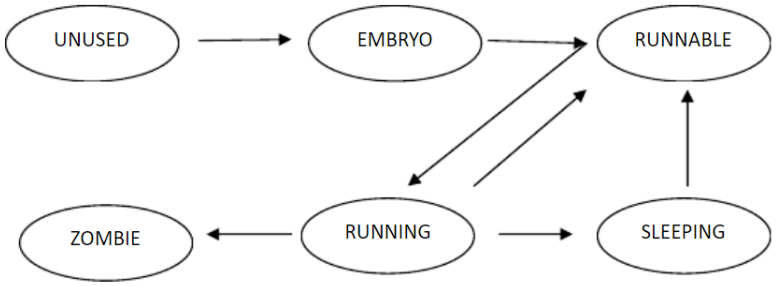
该文件包含一段汇编代码，主要行为是进行上下文切换。具体地，这段代码首先取出栈中传递的参数，其次依次保存%ebp、%edx、%esi、%edi 四个旧的被调用者保存寄存器，然后更换栈指针，弹出新的被调用者保存寄存器，最后通过 ret 指令跳转到新的指令。

2 小结

该部分代码涉及了进程模型的主要内容；同时还包含了部分进程调度的内容。从框架上，与课程介绍是一致的。

该部分代码较为突出的特点有：

- 通过自旋锁保证进程表、调度的安全性。该部分代码中访问进程表往往实现需要获得进程表锁。
- 进程状态的设计与课上介绍的几个模型不同，采用了六状态的模型。该模型具有如下的状态转换：



- `fork` 的机制较为朴素，没有采用写时复制的技术，可能导致性能稍有不足。
- 进程调度的算法较为朴素，总是在进程表中寻找第一个处于 `RUNNABLE` 状态的进程，这可能导致饥饿等问题。
- 其他与进程模型相关的函数中，也总是采用遍历进程表实现，没有采用更高级的数据结构提高效率。当然，xv6 操作系统本身支持的最大进程数仅达 64 个，也相对较少。