Unix V6++创建子进程

同济大学计算机系 操作系统讲义 邓蓉 2023-11-27

一、fork 系统调用

在 UNIX 操作系统中，创建一个新进程的唯一方法就是执行 fork 系统调用[注 1]。执行 fork 系统调用的进程是父进程，创建出的是子进程。

[注 1] 其它操作系统提供有 spawn 系统调用，直接以应用程序为模板创建新进程]

1、fork 系统调用的语义和用法

```
int pid = fork();
```

fork 系统调用成功后，

- 系统中多出了一个和父进程长得一模一样的子进程。它们执行同一个应用程序，对所有信号采用同样的处理方式，拥有相同的变量和变量值，相同的打开文件表、甚至每个文件的读写指针都完全相同……当然它们是“2 个人”，应该有不同的 pid；它们的年龄不一样，新生的子进程所有时间统计量为 0。
- 父进程顺利完成子进程创建任务，fork 系统调用返回，返回值是子进程的 pid（一个正整数）；子进程此时是一个就绪进程。一段时间之后，子进程上台运行，也从 fork 系统调用返回，返回值为 0。

fork 系统调用有 2 种用法。

- 父进程和子进程执行同一个应用程序。fork 系统调用后，子进程与父进程使用同一个共享正文段，但彼此拥有独立的可交换部分。典型的用法如下图，应用程序 example1：

```
int a=0;
main( )
{
    int i ;

    while( ( i=fork( ) ) == -1 );
    if(i)
     {
          a = a+1;
          printf(“parent : a = %d\n”, &a);
     }
    else
    {
          a=a+4;
          printf(“child : a = %d\n”, &a);
```

```
        }
    }
```

代码 1、fork 系统调用的使用

fork 系统调用执行完毕后，父子进程都执行应用程序 example1。

- 父进程将自己的可交换部分复制给子进程，之后父子进程访问自己的数据段和堆栈段。因此子进程数据段中的全局变量 a 与父进程的 a 变量占据不同的物理内存单元；但 fork 返回时刻，它们的值相等，都是 0。局部变量 i 也是如此。
- 父进程从 fork 返回，返回值是子进程的 ID 号，这是一个非负值，所以父进程进 if 分支。子进程也从 fork 返回，返回值是 0，进 else 分支。所以，父子进程分别对自己的私有 a 变量执行加法操作。程序的输出结果 2 行，parent : a = 1 和 child : a = 4。
- 父子进程彼此独立，谁先运行不确定，由调度系统决定。如果父进程先上台运行，该应用程序的输出为：

  parent:a=1

  child:a=4

  如果子进程先上台运行，输出结果为：

  child:a=4

  parent:a=1

- 让子进程转换进程图像（exec）执行另一个程序。

  如果创建子进程是为了让它执行与父进程完全不同的应用程序，子进程 fork 返回后应立即执行 exec 系统调用。称此过程为进程图像转换，下一节详述。

2、fork 系统调用的钩子函数

```
int fork()
{
    int res;
    __asm__ __volatile__ ( "int $0x80":"=a"(res):"a"(2));     // 2#系统调用
    if ( res >= 0 )
        return res;
    return -1;
}
```

代码 2

3、fork 系统调用的入口函数

```
int SystemCall::Sys_Fork()
{
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
    procMgr.Fork();

    return 0;       /* GCC likes it ! */
}
```

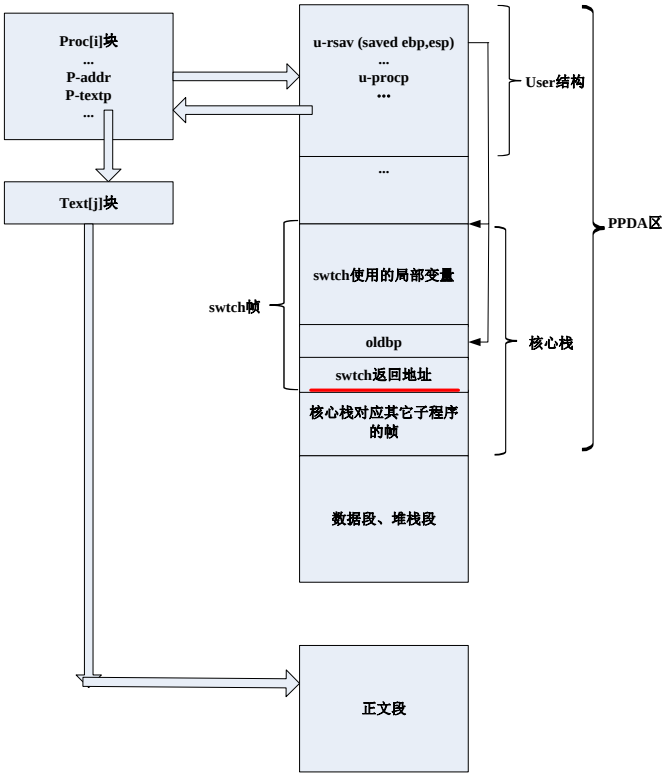
```
void ProcessManager::Fork()
{
    User& u = Kernel::Instance().GetUser();
    Process* child = NULL;;

    /* 为子进程分配空闲的 process 项 */
    for ( int i = 0; i < ProcessManager::NPROC; i++ )
    {
        if ( this->process[i].p_stat == Process::SNULL )
        {
            child = &this->process[i];
            break;
        }
    }
    if ( child == NULL )
    {   /* 没有空闲 process 表项，返回 */
        u.u_error = User::EAGAIN;
        return;
    }

    /* 调用 Newproc( )创建子进程，复制父进程图像 */
L:  if ( this->NewProc() )
    {   /*   新建子进程被 Swtch( )选中上台运行，
         *   执行 Swtch( )函数逻辑，返回值是 1
         *   运行于 Newproc( )栈帧，返回地址是 if 语句。*/
        u.u_ar0[User::EAX] = 0;        // 子进程 fork()系统调用返回 0
        u.u_cstime = 0;                // 清 0 子进程的时间统计量
        u.u_stime = 0;
        u.u_cutime = 0;
        u.u_utime = 0;
    }
    else
    {   /* 子进程图像创建完毕后，父进程 Newproc( )返回 0 */
        u.u_ar0[User::EAX] = child->p_pid; // 父进程 fork()系统调用的返回值是子进程的 pid
    }

    return;
}
```

代码 3

4、NewProc( )创建子进程

新建子进程就绪，可供调度。所以，创建子进程的基本任务就是为它构造一个可供 Swtch 函数调度的就绪态进程图像，其中核心栈顶帧的返回地址指向子进程被 swtch 调度上台后执

行的第 1 条指令。如下图所示。



UNIX 的设计是，让初生的子进程拥有和父进程完全一样的进程图像。为此，需要做的工作包括:

- 为子进程分配一个 Process 结构和一个与所有进程均不相同的 pid
- 子进程共享父进程的共享正文段
- 为子进程的可交换部分分配内存空间
- 复制父进程的可交换部分

UNIX V6++执行上述操作的是函数 ProcessManager::NewProc()。上述操作完成后，父进程创建子进程的工作胜利完成，Newproc 返回；作为现运行进程，父进程完成 fork 系统调用的剩余部分返回用户态，其中最重要的是：令 fork 系统调用的返回值为子进程的 pid。这样父进程就可以执行应用程序的 if 分支（回看代码 1）。

初生的子进程是一个普通的就绪态进程，等待被 Swtch( )调度上台。Swtch( )函数运行在子进程栈顶的 Newproc 帧，所以 Swtch( )返回 fork( )，子进程从标号为 L 的语句开始，实质上做着 fork 系统调用返回的工作。其中最重要地，令 fork 系统调用的返回值为 0，使子进程回到用户态后可以执行应用程序的 else 分支（回看代码 1）。

我们看 NewProc( )函数的代码。强调一下：执行 NewProc()的进程是父进程。代码中出现的 current 指针指向父进程的 Process 结构。我们没有切换 user，代码中出现的 u 是父进程的 user 结构。

```
int ProcessManager::NewProc()
{
    Process* child = 0;
    for (int i = 0; i < ProcessManager::NPROC; i++ )
```

```
    {
        if ( process[i].p_stat == Process::SNULL )    // 为子进程分配空闲的 Process 结构
        {
            child = &process[i];    // child，子进程的 Process 结构
            break;
        }
    }
    if ( !child )
    {
        Utility::Panic("No Proc Entry!");    // 无空闲 Process 结构，系统崩溃，等待人工干预
    }

    User& u = Kernel::Instance().GetUser();    // 父进程的 user
    Process* current = (Process*)u.u_procp;    // 父进程的 Process 结构
    current->Clone(*child);    // 复制父进程 Process 结构，资源引用计数器++

    SaveU(u.u_rsav);    // 在 user 结构（字段 u_rsav）中保存 Newproc( )栈帧的顶部和基地址

    /* pgTable 暂存父进程相对虚实地址映射表的起始地址 */
    PageTable* pgTable = u.u_MemoryDescriptor.m_UserPageTableArray;
    u.u_MemoryDescriptor.Initialize();    //为父进程分配新的相对虚实地址映射表

    if ( NULL != pgTable )
    {    // 把原先的相对表复制过来
        u.u_MemoryDescriptor.Initialize();
        Utility::MemCopy((unsigned long)pgTable, (unsigned long) u.u_MemoryDescriptor.
m_UserPageTableArray, sizeof(PageTable) * MemoryDescriptor::USER_SPACE_PAGE_TABLE_CNT);
    }

    u.u_procp = child;    // 令父进程的 u_procp 指向子进程的 Process 块。这样，复制给子进程的
u_procp 指针将指向子进程的 Process 块。

    /* 为子进程的可交换部分分配内存空间 */
    UserPageManager& userPageManager = Kernel::Instance().GetUserPageManager();
    unsigned long srcAddress = current->p_addr;
    unsigned long desAddress = userPageManager.AllocMemory(current->p_size);
    if ( desAddress == 0 )
    {    // 内存不够，将父进程的可交换部分复制一份到盘交换区。这个复本是子进程的可交换部分
        current->p_stat = Process::SIDL;    // 父进程置 SIDL 标记，复制过程中不能被调度
        child->p_addr = current->p_addr;    // 父子进程共用内存中的一份可交换部分
        SaveU(u.u_ssav);    //在 user 结构（字段 u_ssav）中再次保存 Newproc( )栈帧的顶部和基地址
        this->XSwap(child, false, 0);    // 复制父进程的可交换部分。最后一个参数是 0，内存中的
图像不释放。父进程用内存中的可交换部分，子进程用盘交换区上的可交换部分。
```

```
            child->p_flag |= Process::SSWAP;     // 进程图像复制完成后，子进程置 SSWAP 标识，引导 Swtch，从 u_ssav 数组中恢复栈顶帧。
            current->p_stat = Process::SRUN; //父进程恢复 SRUN，这就清除了临时设置的 SIDL 标识
        }
        else
        {   // 内存分配成功，将父进程的可交换部分复制给子进程
            child->p_addr = desAddress;         // 登记子进程可交换部分的起始地址
            int n = current->p_size;
            while (n--)
            {
                Utility::CopySeg(srcAddress++, desAddress++);   //内存复制操作，一次一个字节
            }
        }

        u.u_procp = current;   // 先前为了复制方便（让子进程 user 结构的 u_procp 指向其 Process 结构 child），我们暂时让父进程的 u_procp 指向 child。现在改回来。
        u.u_MemoryDescriptor.m_UserPageTableArray = pgTable;     // 先前为复制方便（让子进程 user 结构中相对表指针 u_MemoryDescriptor.m_UserPageTableArray 指向子进程的相对表），我们暂时让父进程的相对表指针指向子进程相对表的起始地址。现在改回来。

        return 0;   //父进程 Newproc 返回 0
}
```

代码 4

辅助函数 1:

```
void Process::Clone(Process& proc)   // 指针 this 和 proc 分别指向父进程和子进程的 Process 结构
{
        User& u = Kernel::Instance().GetUser();

    /* 拷贝父进程 Process 结构中的大部分数据 */
        proc.p_size = this->p_size;
        proc.p_stat = Process::SRUN;
        proc.p_flag = Process::SLOAD;
        proc.p_uid = this->p_uid;
        proc.p_ttyp = this->p_ttyp;
        proc.p_nice = this->p_nice;
        proc.p_textp = this->p_textp;

        proc.p_pid = ProcessManager::NextUniquePid();   // 为子进程分配 pid
        proc.p_ppid = this->p_pid;     // 子进程登记父进程的 pid

        proc.p_pri = 0;      // 确保 child 的优先数较小，先于父进程运行（Unix V6++并无此必要
        proc.p_time = 0;   // 驻留时间清 0
```

```
    /* 父进程已打开的文件，子进程无需再打开，这一点复制进程可交换部分的时候已经完成了。这里
我们需要对父进程引用的所有 File 结构，引用计数+1 */
    for ( int i = 0; i < OpenFiles::NOFILES; i++ )
    {
        File* pFile;
        if ( (pFile = u.u_ofiles.GetF(i)) != NULL )
        {
            pFile->f_count++;
        }
    }

    /*
    * GetF()访问 u.u_ofiles 中的空闲项会产生出错码，如不清除将导致 fork 系统调用失败。
    */
    u.u_error = User::NOERROR;

    /* 正文段引用计数++ */
    if ( proc.p_textp != 0 )
    {
        proc.p_textp->x_count++;
        proc.p_textp->x_ccount++;
    }

    /* 当前工作目录引用计数++ */
    u.u_cdir->i_count++;
}
```

辅助函数 4:

ProcessManager.cpp

```
unsigned int ProcessManager::m_NextUniquePid = 0;


unsigned int ProcessManager::NextUniquePid()
{
    return ProcessManager::m_NextUniquePid++;
}
```

辅助函数 3:

```
void Utility::CopySeg(unsigned long src, unsigned long des)
{
    PageTableEntry* PageTable = Machine::Instance().GetKernelPageTable().m_Entrys;

    /* 映射需要借用父进程的 2 个 PTE，一个映射地址为 src 的字节，一个映射地址为 des 的字节 */
    unsigned long oriEntry1 = PageTable[borrowedPTE].m_PageBaseAddress;
```

```
    unsigned long oriEntry2 = PageTable[borrowedPTE + 1].m_PageBaseAddress;

    /* 分别将 src 字节和 des 字节所在的物理页框号填入借来的 PTE */
    PageTable[borrowedPTE].m_PageBaseAddress = src / PageManager::PAGE_SIZE;
    PageTable[borrowedPTE+1].m_PageBaseAddress = des / PageManager::PAGE_SIZE;

    /* 计算 src 字节和 des 字节的虚地址 */
    unsigned char* addressSrc = (unsigned char*)(0xC0000000 +
        borrowedPTE*PageManager::PAGE_SIZE + src % PageManager::PAGE_SIZE);
    unsigned char* addressDes = (unsigned char*)(0xC0000000 +
        (borrowedPTE + 1)*PageManager::PAGE_SIZE + des % PageManager::PAGE_SIZE);
    FlushPageDirectory();   //要刷新 PTE

    *addressDes = *addressSrc;   // 复制一个字节

    /*   借来的 PTE 恢复原先的映射关系 */
    PageTable[borrowedPTE].m_PageBaseAddress = oriEntry1;
    PageTable[(borrowedPTE + 1)].m_PageBaseAddress = oriEntry2;
    FlushPageDirectory();   //要刷新 PTE
}
```