基础实验篇第4章

多进程与多线程

上一节的问题：处理大规模文件中的字符串过滤

读取与字符串查找是交替执行的

如何更高效地处理这个问题?

- 只有一个CPU的单处理器上，在同一时间只能实际执行一个任务
- 若等待读文件的数据到达再处理，会导致任务阻塞，浪费时间
- 同时，查找计算的过程中，存储设备也在等待
- 现代处理器都是多核心的，可以并行地执行多个任务
- 如何充分利用这个优势提升程序效率?

- **并行化处理->多进程和多线程**

1 Linux的调度机制

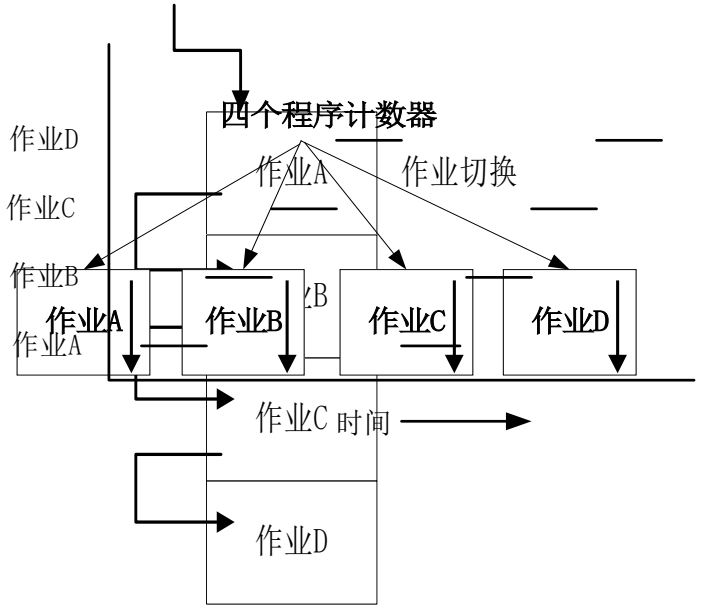
2 多进程协作的编程模式

3 多线程协作的编程模式

4 两种方法的优劣

Linux的调度机制

分时复用机制

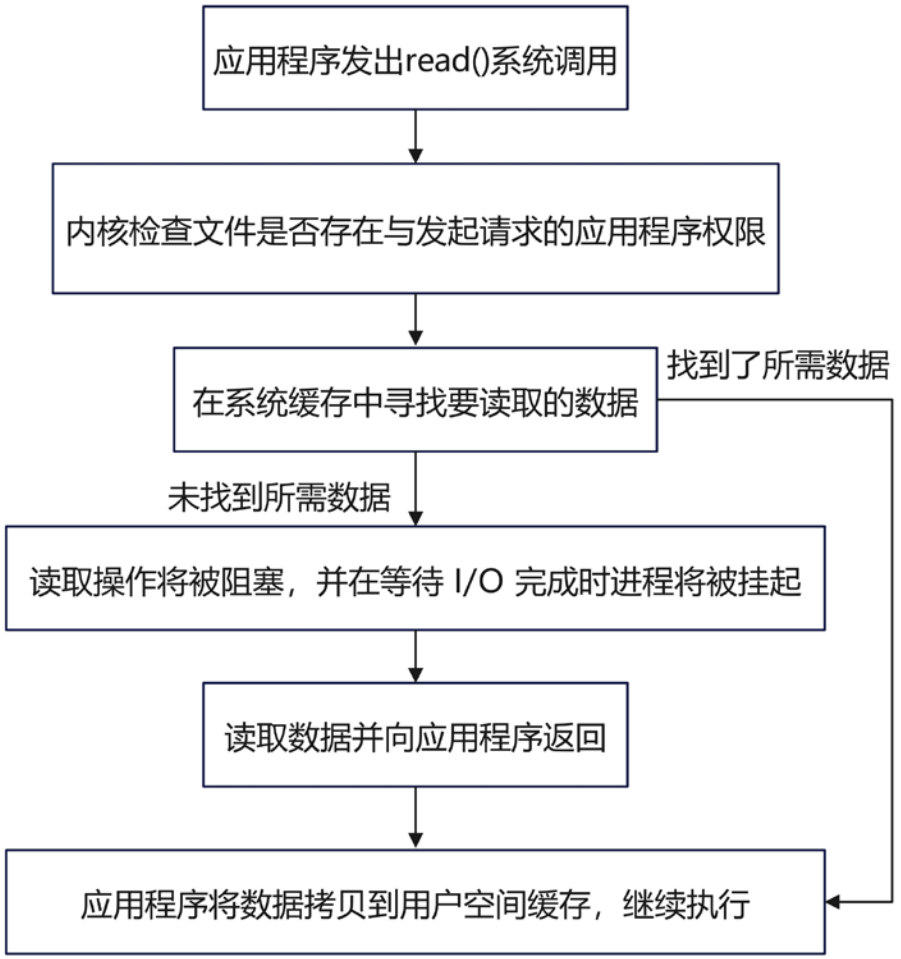


在操作系统眼中，同一时刻仅处理一个任务

操作系统同时接纳了多个任务，并且在宏观层面实现了任务的并行

分时复用：同一时刻仅处理一个任务，但多个任务交替执行

阻塞式系统调用——以read()为例

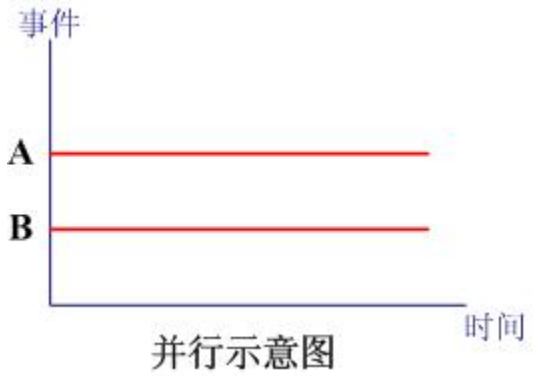


- 常用于I/O请求
- 优点：简单、可靠
- 缺点：响应时间延长、占用资源

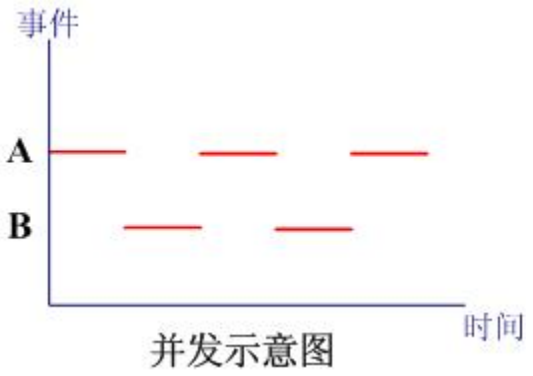
在上一节实验中，采用先读取文件再查找的方式，读取时会造成进程的阻塞，对性能影响很大。

如何高效处理？→ 并行提升效率

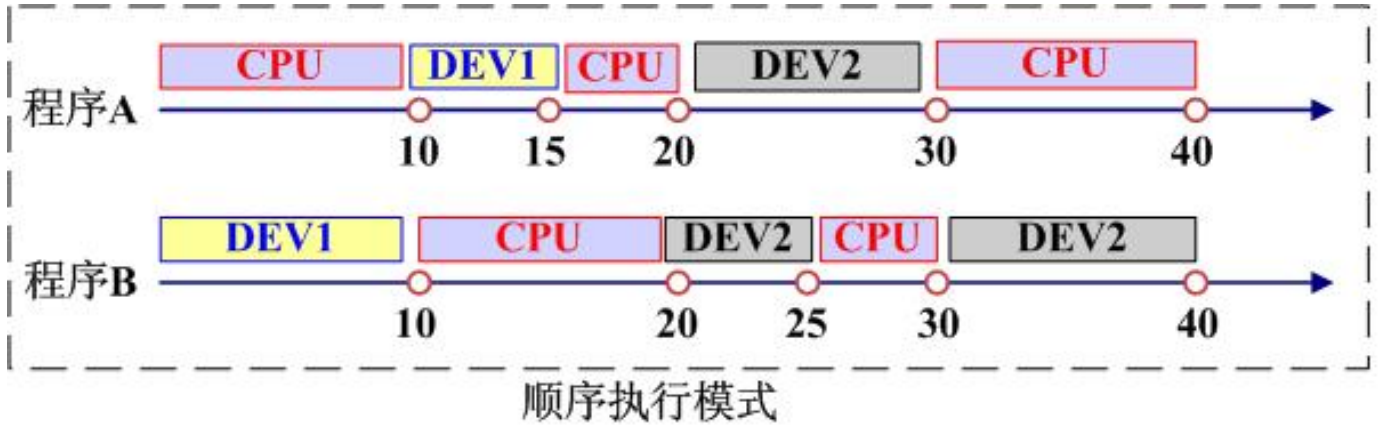
并发执行得到的并行效果



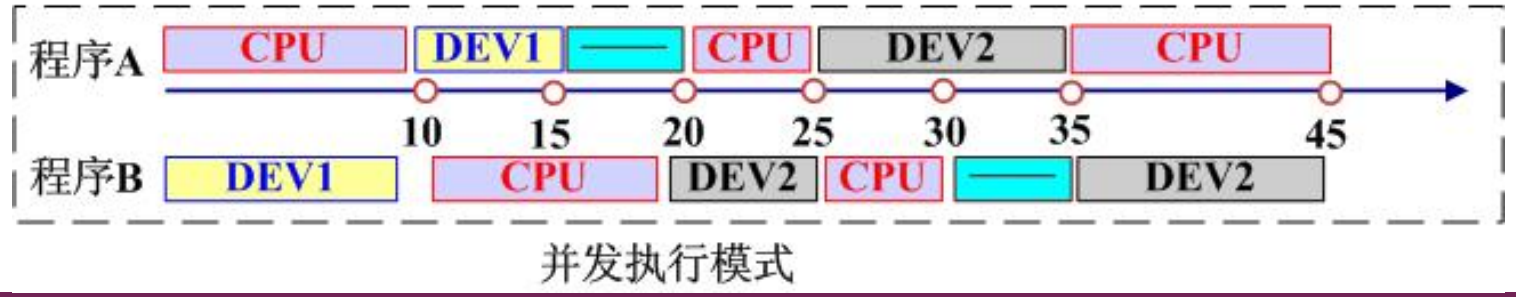
并行示意图



并发示意图



顺序执行模式

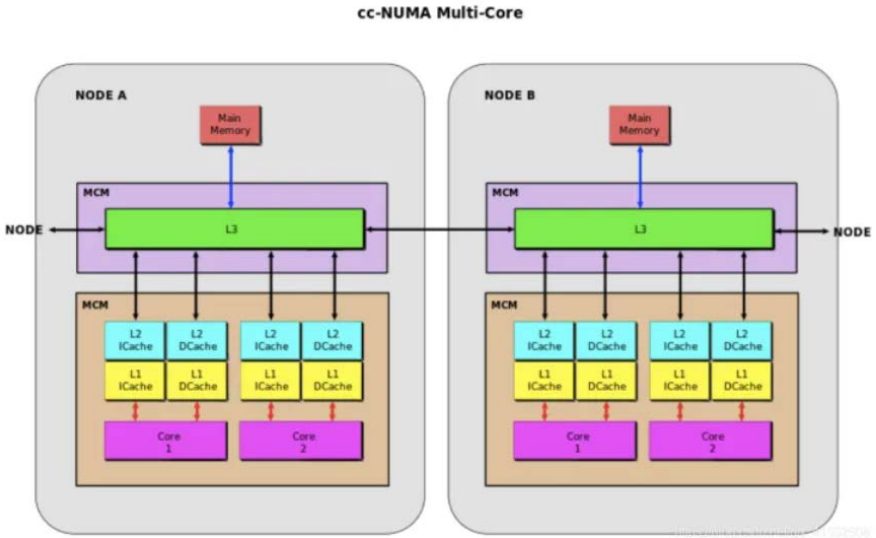


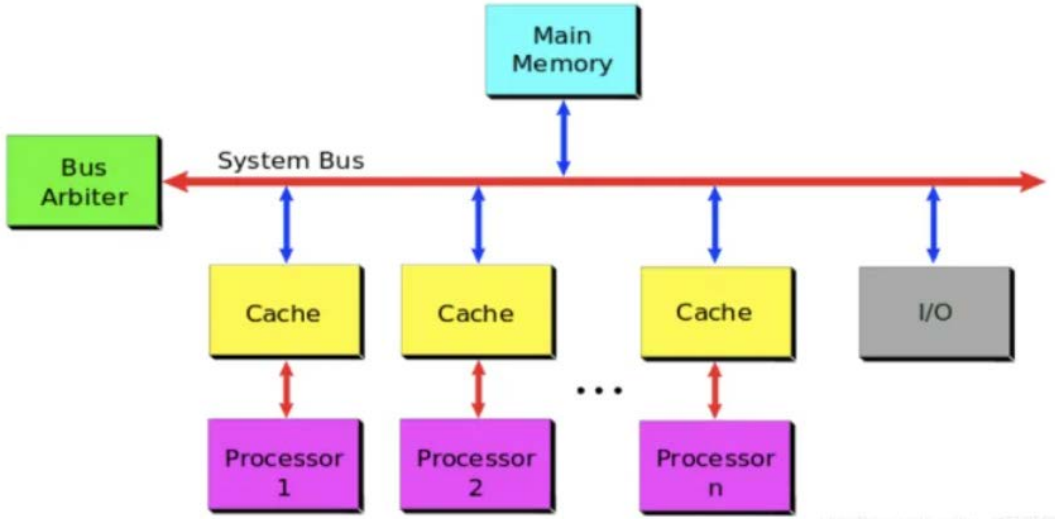
并发执行模式

单个程序的运行无法完全发挥CPU和设备的性能

多核心任务调度

- 并行：在同一个时刻，存在两个以上任务在同时运行。
- 多核心处理器可以将工作分散到多个cpu上，cpu资源越多就运行越快，达到并行
- 充分利用这些核心的方式就是尽可以的产生可以并行执行的任务





- 多核：单个 CPU 中有多个可执行单元
- 多处理器：拥有多个 CPU，每个 CPU 中也可能有多个核

多进程协作的编程模式

如何添加多个进程——创建子进程

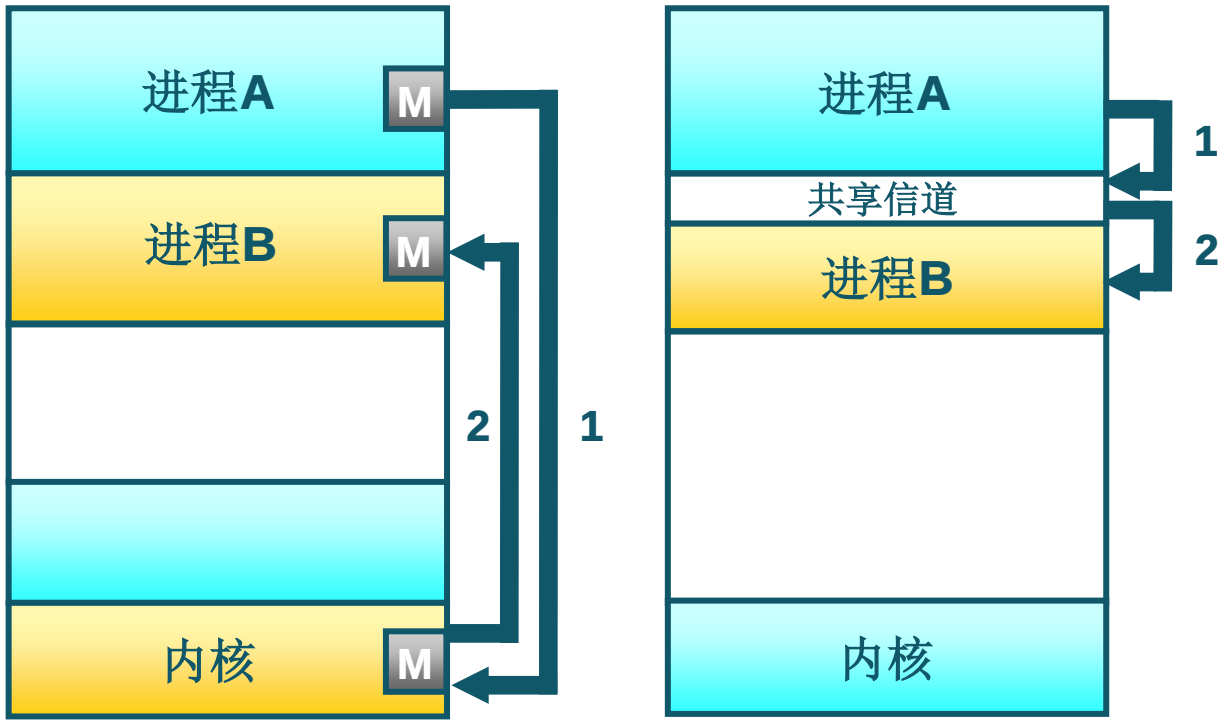
```
1.      pid_t pid1, pid2;
2.      int status1, status2;

3.      if ((pid1 = fork()) == 0) {
4.          // 子进程1，查找文件夹f1中的所有包含特殊字符的文件
5.          execl("./mmap", "mmap","f1", NULL);
6.      } else if ((pid2 = fork()) == 0) {
7.          // 子进程2，查找文件夹f2中的所有包含特殊字符的文件
8.          execl("./mmap", "mmap","f2", NULL);
9.      } else {
10.         // 父进程阻塞等待两个子进程结束
11.         waitpid(pid1, &status1, 0);
12.         waitpid(pid2, &status2, 0);
13.     }
```

- 使用**fork()**函数创建两个子进程
- 使用**execl()**函数让他们执行不同任务
- mmap是一个独立的可执行程序，用于查找一个文件夹中所有的特殊字符串，f1和f2是文件夹名，作为该程序的参数
- 使用**wait()**函数让父进程阻塞等待两个子进程结束

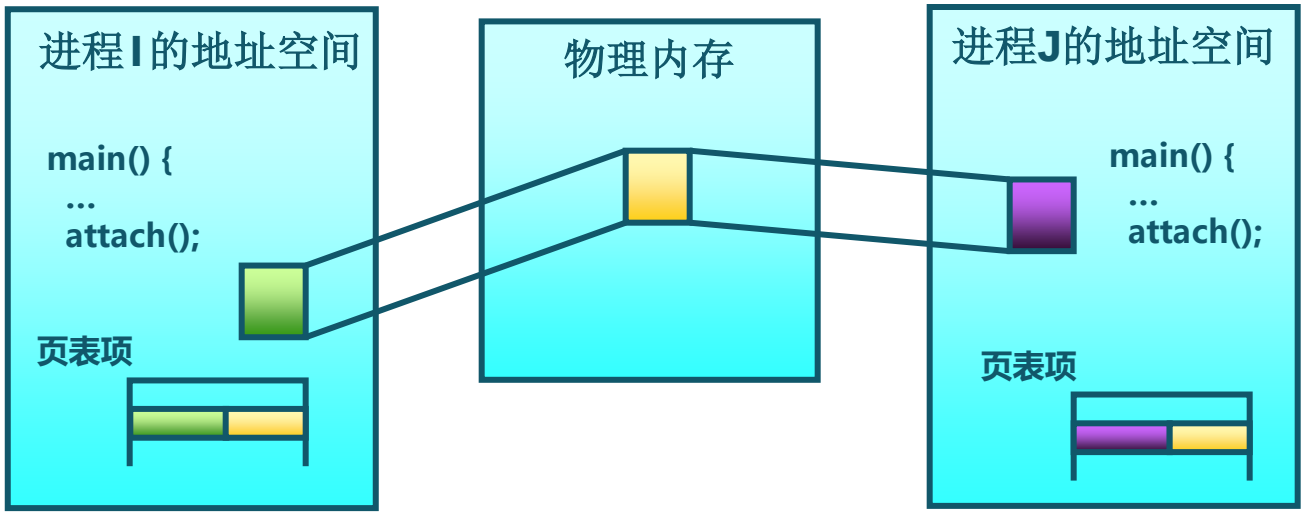
问题：单个进程结束任务后，如何汇集合并最终结果?

进程间通信的方式



间接通信　　直接通信

共享内存的实现



- 最快的方法
- 一个进程写另外一个进程立即可见
- 没有系统调用干预
- 没有数据复制
- 不提供同步
  - 由程序员提供同步

多个进程间通信——创建共享内存

```
1.    int shmid;
2.    int *shmaddr;
3.    int i;
4.    pid_t pid;
5.    // 创建共享内存
6.    shmid = shmget(IPC_PRIVATE, sizeof(int), 0666);
7.    if (shmid < 0) {
8.        perror("shmget error");
9.        exit(1);
10.   }
11.   // 将共享内存映射到当前进程的地址空间
12.   shmaddr = (int *)shmat(shmid, NULL, 0);
13.   if (shmaddr == (int *)-1) {
14.       perror("shmat error");
15.       exit(1);
16.   }
17.   // 将共享内存初始化为0
18.   *shmaddr = 0;
```

- 共享内存用于实现多个进程的通信。
- shmget用于创建虚拟内存
  - 第一个参数为内存的key
  - 返回值是内存的id
- shmat将共享内存接入地址空间
- 特殊字符串的计算结果将会累加到共享内存。

获取子进程计数结果

```
1.  // 获取子进程的返回值，也就是该文件夹中符合条件的计数结果
2.      *shmaddr+=WEXITSTATUS(status1);
3.      *shmaddr+=WEXITSTATUS(status2);

4.      // 输出文件夹中文件的总行数
5.      printf("Total count: %d\n", *shmaddr);

6.      // 删除共享内存
7.      shmctl(shmid, IPC_RMID, NULL);
8.      return 0;
```

- 使用WEXITSTATUS宏获取子进程返回值。
- 将返回值写入共享内存。
- 删除共享内存。

查看共享内存的使用

ipcs命令是一个Linux系统下的进程间通信工具，用于显示和控制共享内存、消息队列和信号量等IPC资源的状态。

ipcs命令的常用选项包括:

- -a：显示所有IPC资源的详细信息。
- -m：显示共享内存的详细信息。
- -q：显示消息队列的详细信息。
- -s：显示信号量的详细信息。
- -p：显示IPC资源的创建者和最后修改者等进程信息。
- -u：显示当前用户的IPC资源使用情况。

```
haruka@ubuntu:~/Documents/OSbook/lab_basic/lab4/multiprocess$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 9          haruka     600        524288     2          dest
0x00000000 13         haruka     600        524288     2          dest
0x00000000 30         haruka     600        524288     2          dest
```

未回收共享内存的影响

- 共享内存是系统级资源
- 共享内存无法获得使用共享内存的其他进程的信息，不会自动释放
- 如果不手动删除共享内存，可能会导致内存泄漏、资源浪费等问题

执行程序前：

```
haruka@ubuntu:~/Documents/OSbook/lab_basic/lab4/multiprocess$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 9          haruka     600        524288     2          dest
0x00000000 13         haruka     600        524288     2          dest
```

执行程序后（未在程序中手动删除共享内存）：

```
haruka@ubuntu:~/Documents/OSbook/lab_basic/lab4/multiprocess$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 9          haruka     600        524288     2          dest
0x00000000 13         haruka     600        524288     2          dest
0x00000000 33         haruka     666        4          0
```

如何手动删除共享内存

1.使用ipcrm命令

ipcrm命令是用于删除系统中的IPC资源（Inter-Process Communication，进程间通信）的命令。IPC资源包括共享内存、消息队列和信号量。

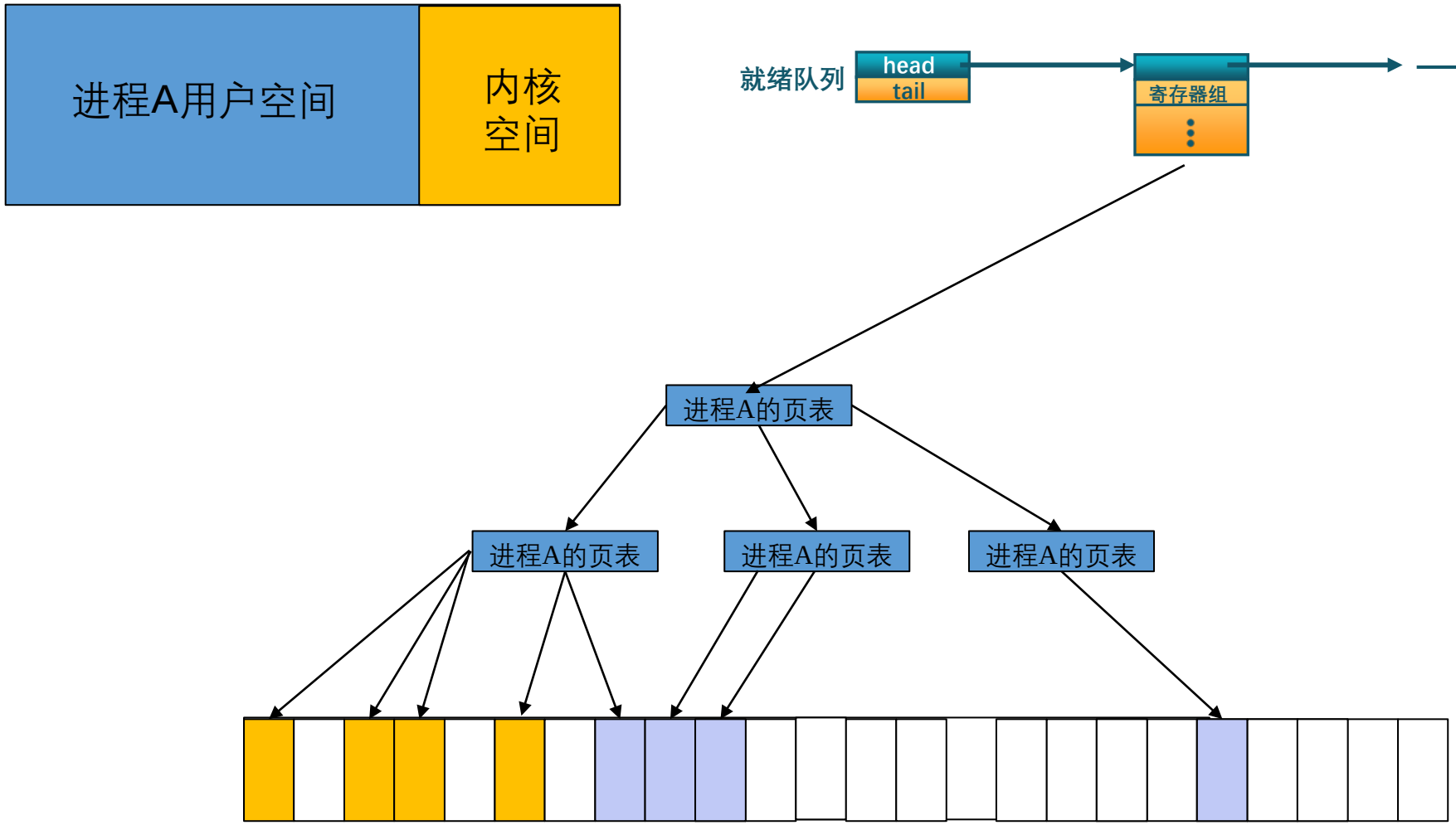
常用的选项包括:

- -m：删除共享内存段。
- -q：删除消息队列。
- -s：删除信号量。

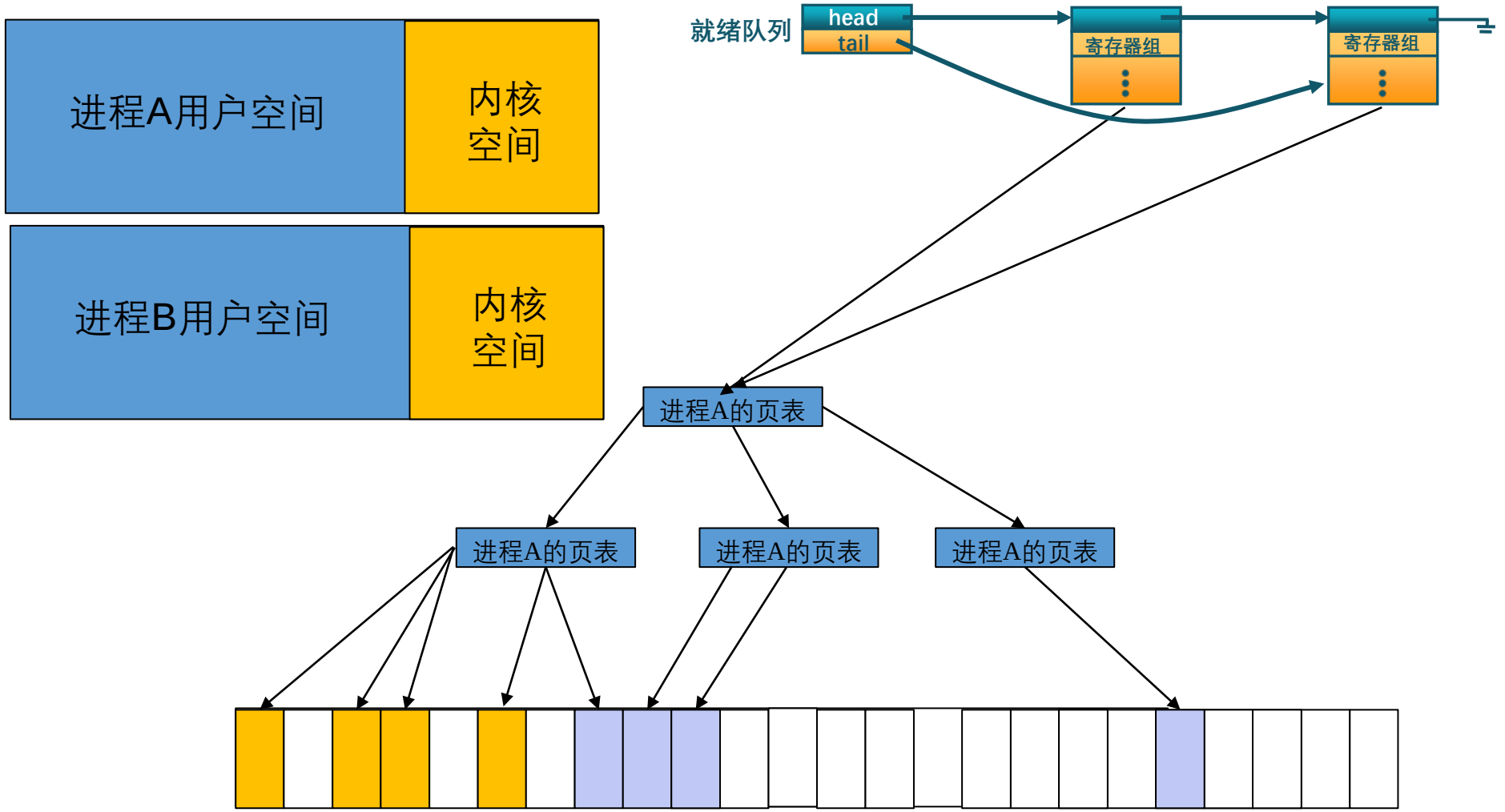
资源ID是要删除的IPC资源的标识符，可以是IPC键值或IPC标识符。

2.重启

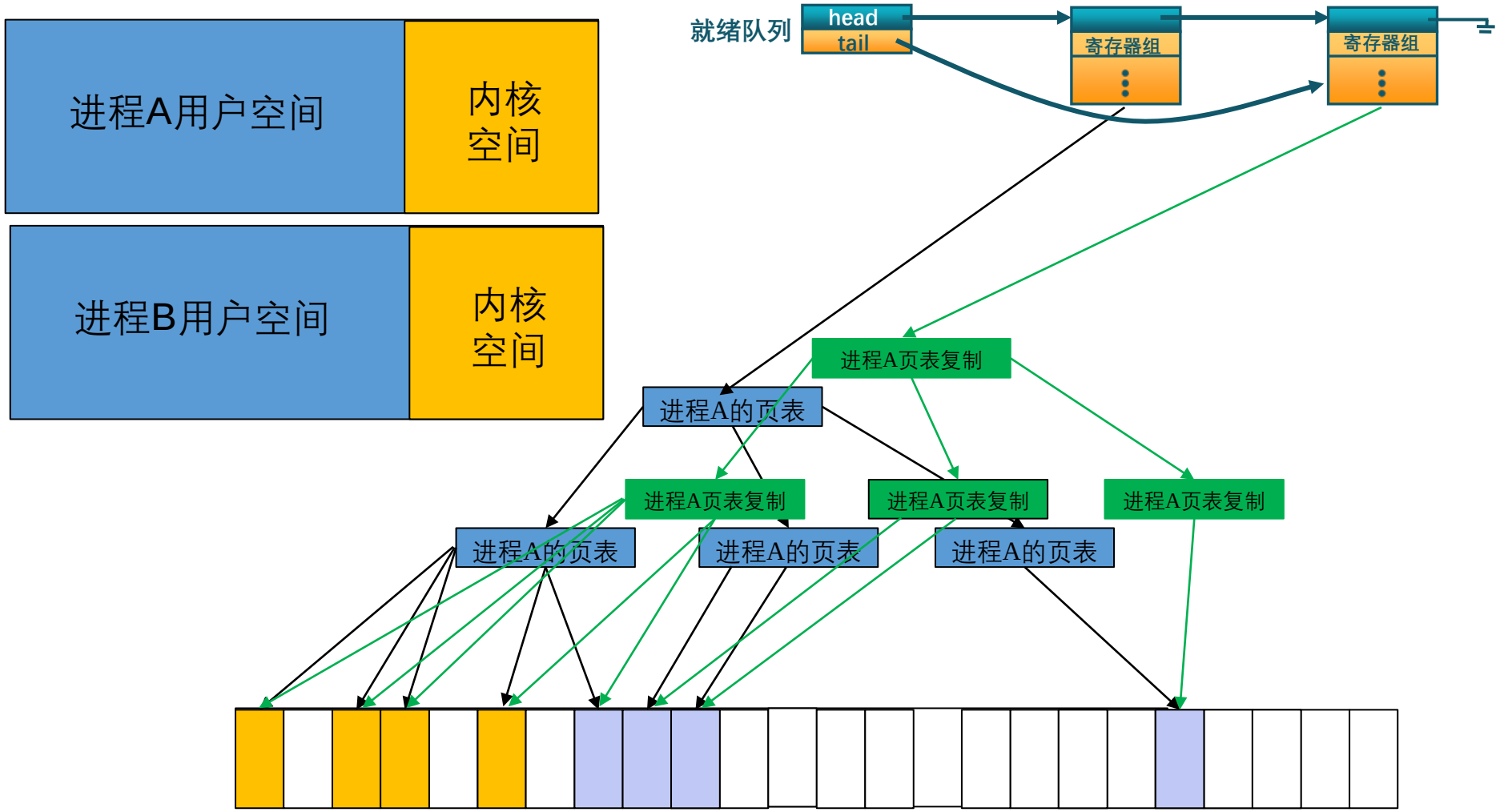
创建多个进程的原理


进程A用户空间
内核空间
就绪队列
head
tail
寄存器组
进程A的页表
进程A的页表
进程A的页表
进程A的页表

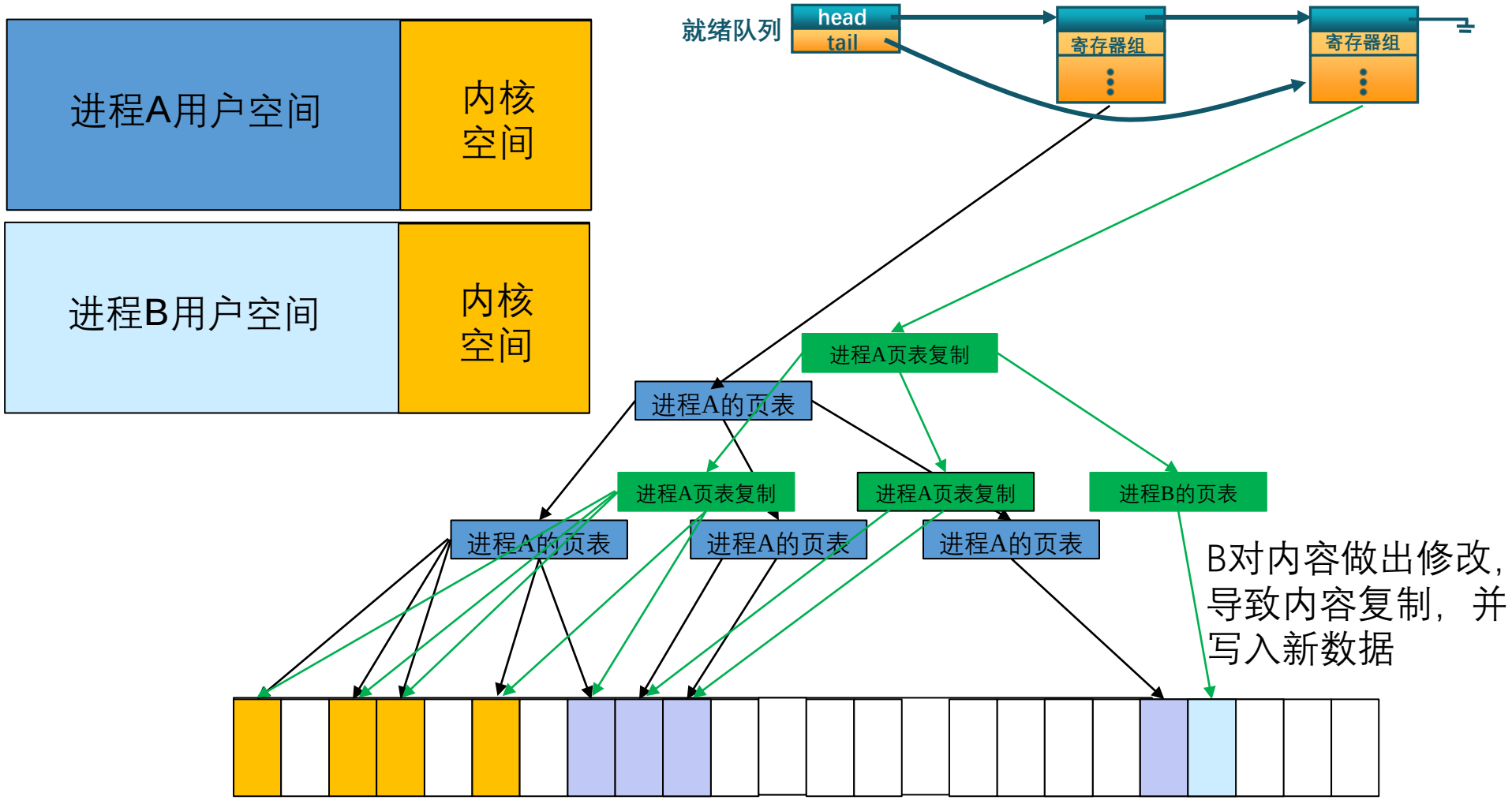
创建多个进程的原理


进程A用户空间
内核空间
进程B用户空间
内核空间
就绪队列
head
tail
寄存器组
寄存器组
进程A的页表
进程A的页表
进程A的页表
进程A的页表

创建多个进程的原理


进程A用户空间
内核空间
进程B用户空间
内核空间
就绪队列
head
tail
寄存器组
寄存器组
进程A页表复制
进程A的页表
进程A页表复制
进程A页表复制
进程A页表复制
进程A的页表
进程A的页表
进程A的页表

创建多个进程的原理


进程A用户空间
内核空间
进程B用户空间
内核空间
就绪队列
head
tail
寄存器组
寄存器组
进程A页表复制
进程A的页表
进程A页表复制
进程A页表复制
进程B的页表
进程A的页表
进程A的页表
进程A的页表
B对内容做出修改，导致内容复制，并写入新数据

多进程会不会占用更多的内存?

运行程序前:

```
Every 0.1s: free                                ubuntu: Fri Oct 13 06:03:55 2023

               total        used        free      shared  buff/cache   available
Mem:         4002248     1011488     2075460        2632      915300     2738360
Swap:        2097148           0     2097148
```

运行程序后:

```
Every 0.1s: free                                ubuntu: Fri Oct 13 06:04:07 2023

               total        used        free      shared  buff/cache   available
Mem:         4002248     1011800     2075136        2636      915312     2738044
Swap:        2097148           0     2097148
```

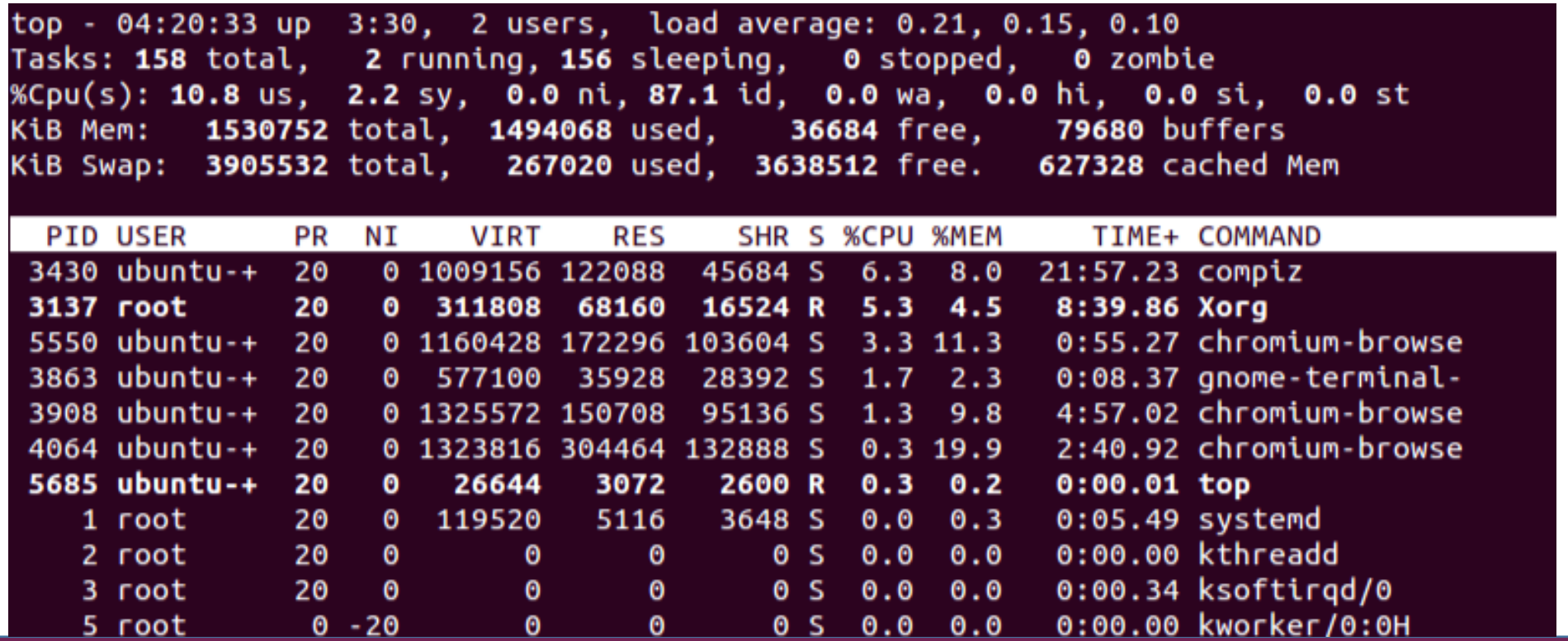
- free是Linux系统的内存监控工具，用于查看系统中可用的内存的使用情况
- 执行watch -n 0.1 free，查看每隔0.1s的内存变化
- 每个进程都是由原来的进程fork，然后exec mmap命令，因此会占用更多内存

top命令

top命令经常用来监控linux的系统状况，是常用的性能分析工具，能够实时显示系统中各个进程的资源占用情况。

使用方式 top [-d number] | top [-bnp]

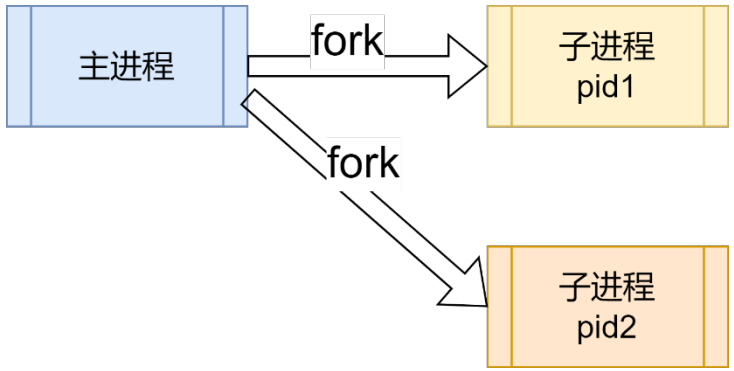
- -d：number代表秒数，表示top命令显示的页面更新一次的间隔，默认5秒。
- -b：以批次的方式执行top。
- -n：与-b配合使用，表示需要进行几次top命令的输出结果。
- -p：指定特定的pid进程号进行观察。
-
-



如何添加多个进程——创建子进程

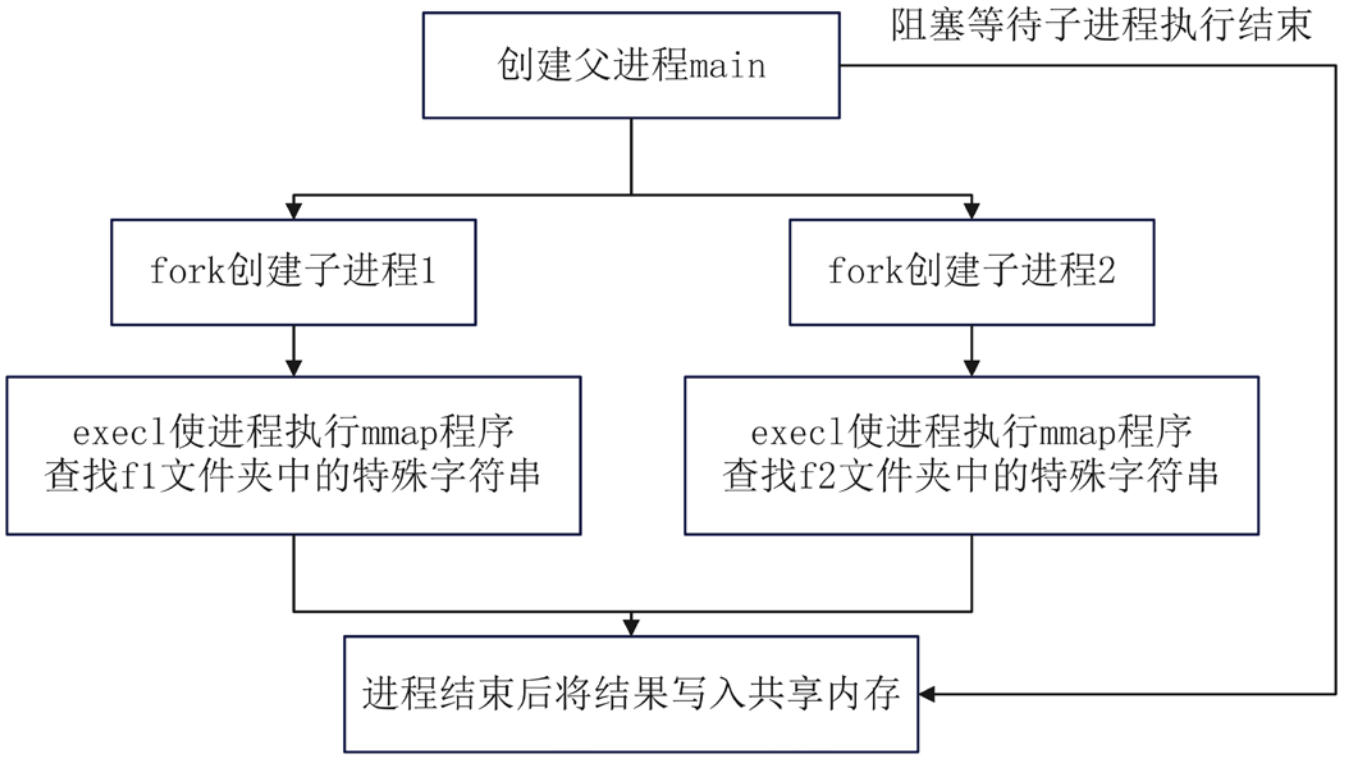
```
1.      pid_t pid1, pid2;
2.      int status1, status2;

3.      if ((pid1 = fork()) == 0) {
4.          // 子进程1，查找文件夹f1中的所有包含特殊字符的文件
5.          execl("./mmap", "mmap","f1", NULL);
6.      } else if ((pid2 = fork()) == 0) {
7.          // 子进程2，查找文件夹f2中的所有包含特殊字符的文件
8.          execl("./mmap", "mmap","f2", NULL);
9.      } else {
10.         // 父进程阻塞等待两个子进程结束
11.         waitpid(pid1, &status1, 0);
12.         waitpid(pid2, &status2, 0);
13.     }
```



虽然两个子进程内容几乎相同，但是它们没有父子关系，因此无法合并内容

程序执行流程



缺点：共享内存创建管理逻辑控制复杂，占用内存多

解决方案：使用多线程编程模式，数据无需处于隔离的地址空间

多线程协作的编程模式

创建线程执行的函数——用于匹配字符串

```
1.  FILE *fp;
2.  char *pattern = "^From ilug-admin@linux.ie.*Aug.*";
3.  int total_count;
4.  void *thread_func(void *arg)
5.  {
6.      char buf[1024];
7.      int thread_num = *(int *)arg;

8.      while (fgets(buf, 1024, fp) != NULL)
9.      {
10.         // 匹配敏感字符串
11.         .....
12.       }
13.     pthread_exit(NULL);
14. }
```

- **thread_func**即创建新线程需要执行的函数，实现特殊字符串的查找。
- 全局变量**total_count**用于存储计数结果，每查找到一个特殊字符串，就将它增加1，免去了共享内存的使用

创建线程执行的函数——用于匹配字符串

```
1.  FILE *fp;
2.  char *pattern = "^From ilug-admin@linux.ie.*Aug.*";
3.  int total_count;
4.  void *thread_func(void *arg)
5.  {
6.      char buf[1024];
7.      int thread_num = *(int *)arg;

8.      while (fgets(buf, 1024, fp) != NULL)
9.      {
10.         // 匹配敏感字符串
11.         .....
12.       }
13.     pthread_exit(NULL);
14. }
```

- **thread_func**即创建新线程需要执行的函数，实现特殊字符串的查找。
- 全局变量**total_count**用于存储计数结果，每查找到一个特殊字符串，就将它增加1，免去了共享内存的使用

创建多个线程共同读取文件

```
1.  // 创建多线程
2.      for (i = 0; i < MAX_THREAD_NUM; i++)
3.      {
4.          thread_num[i] = i;
5.          ret = pthread_create(&tid[i], NULL, thread_func, &thread_num[i]);
6.          if (ret != 0)
7.          {
8.              printf("Create thread %d failed\n", i);
9.              exit(1);
10.         }
11.     }

12.     // 分离线程
13.     for (i = 0; i < MAX_THREAD_NUM; i++)
14.     {
15.         pthread_join(tid[i], NULL);
16.     }
```

- **pthread_create**用于创建新线程。
- **pthread_join**用于用于等待一个线程的结束， 并接收它的返回值。

如何查看一个进程中的多个线程

- **ps -eLf：**列出系统中所有进程及其对应的线程信息
- PID：进程号 PPID：父进程号 LWP：线程号

```
a@ubuntu:~/Documents/OSbook/lab_basic/lab4/multiprocess$ ps -eLF
    PID    PPID     LWP  C NLWP    SZ   RSS PSR STIME TTY          TIME CMD
      1       0       1  0    1 41972 11740   0 05:25 ?        00:00:01 /sbin/init auto nopro
      2       0       2  0    1     0     0   0 05:25 ?        00:00:00 [kthreadd]
      3       2       3  0    1     0     0   0 05:25 ?        00:00:00 [rcu_gp]
      4       2       4  0    1     0     0   0 05:25 ?        00:00:00 [rcu_par_gp]
      6       2       6  0    1     0     0   0 05:25 ?        00:00:00 [kworker/0:0H-kblockd
      9       2       9  0    1     0     0   0 05:25 ?        00:00:00 [mm_percpu_wq]
     10       2      10  0    1     0     0   0 05:25 ?        00:00:00 [ksoftirqd/0]
     11       2      11  0    1     0     0   1 05:25 ?        00:00:01 [rcu_sched]
     12       2      12  0    1     0     0   0 05:25 ?        00:00:00 [migration/0]
     13       2      13  0    1     0     0   0 05:25 ?        00:00:00 [idle_inject/0]
     14       2      14  0    1     0     0   0 05:25 ?        00:00:00 [cpuhp/0]
```

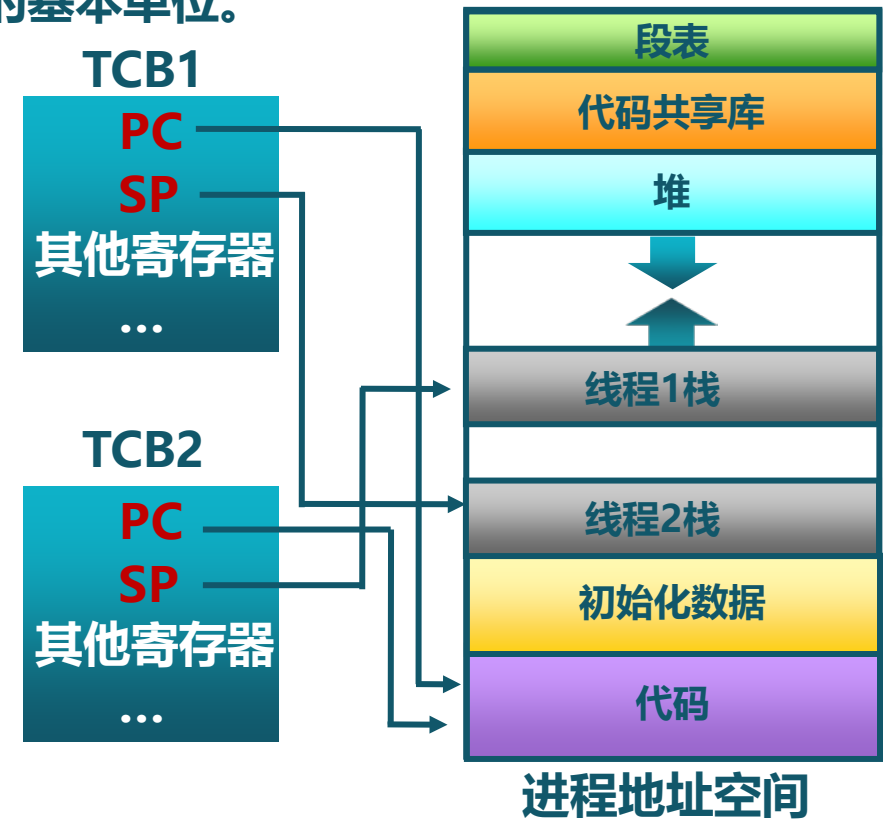
- **ps -eLF | grep 'process id'：**用于查看指定进程的多个线程

```
haruka@ubuntu:~/Documents/OSbook/lab_basic/lab4/multiprocess$ ps -eLF | grep 911
root         911       1     911  0    3 59776 12120   1 05:25 ?        00:00:00 /usr/lib/policykit-1/polkitd --no-debug
root         911       1     917  0    3 59776 12120   0 05:25 ?        00:00:00 /usr/lib/policykit-1/polkitd --no-debug
root         911       1     961  0    3 59776 12120   1 05:25 ?        00:00:00 /usr/lib/policykit-1/polkitd --no-debug
```

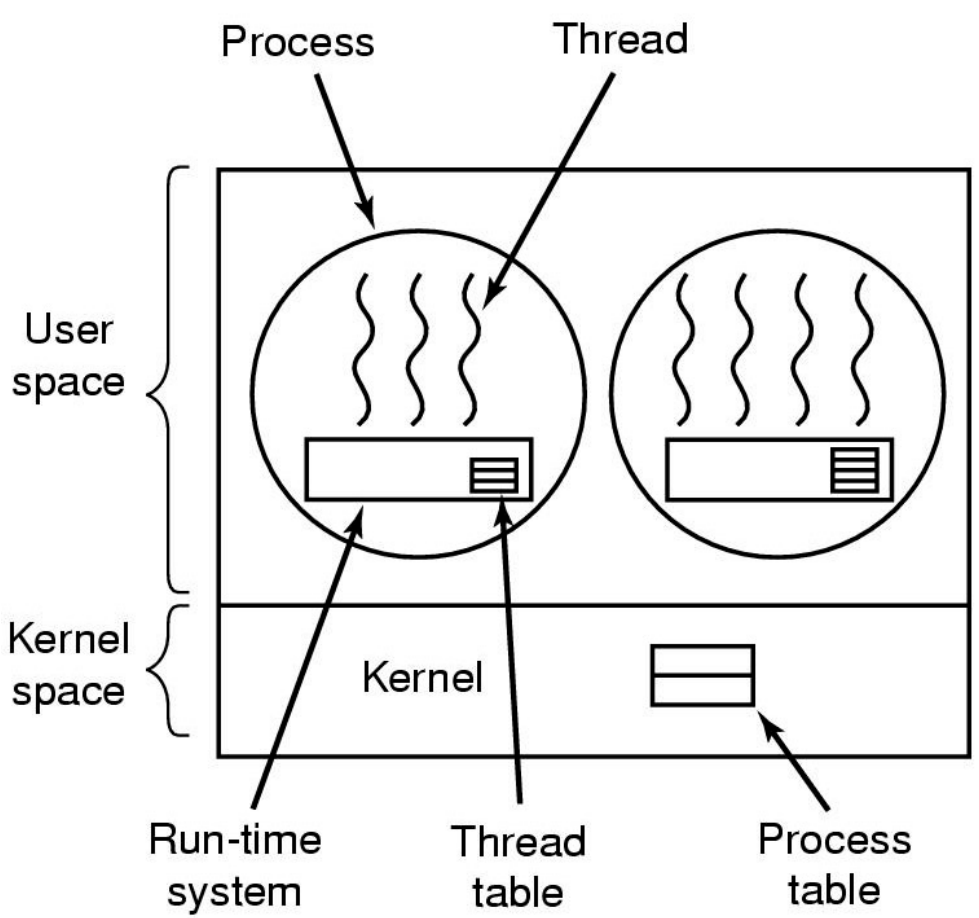
线程的概念

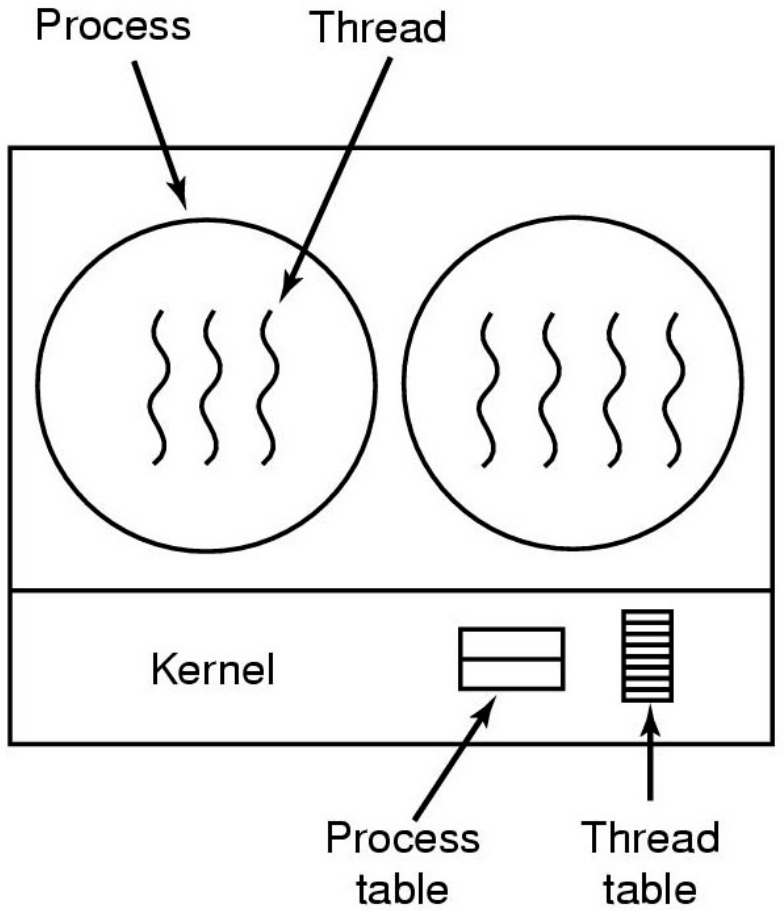
线程是进程的一部分，描述指令流执行状态。它是进程中的指令执行流的最小单元，是CPU调度的基本单位。

- 进程的资源分配角色：进程由一组相关资源构成，包括地址空间（代码段、数据段）、打开的文件等各种资源
- 线程的处理机调度角色：线程描述在进程资源环境中的指令流执行状态



用户态线程（ULT）vs内核级线程（KLT）





用户级线程（ULT）vs内核级线程（KLT）

ULT管理优势:

- 线程切换不需要内核模式特权.
- 线程调用可以是应用程序级的,根据需要可改变调度算法,但不会影响底层的操作系统调度程序.
- ULT管理模式可以在任何操作系统中运行,不需要修改系统内核,线程库是提供应用的实用程序。

ULT的劣势:

- 系统调用会引起进程阻塞
- 这种线程不利于使用多处理器并行

核心级线程（KLT）：线程由OS内核进行管理，内核给应用程序级提供系统调用，实现对线程的使用。

特点:

- 线程在内核中有保存的信息,系统调度是基于线程完成的.
- 可克服ULT的两个缺点,且内核程序本身也可以是多线程结构的.

劣势:

- 线程间的控制转换需要转换到内核模式.（为什么）

轻量级线程

• 轻量级线程：由用户程序创建和管理的线程，但在运行时可以映射到内核级线程上。兼顾用户级线程和内核级线程的优点，既能提供灵活性，又能提供较好的性能。

• 由用户程序创建和管理：用户程序可以自己决定线程的创建、调度和销毁，而不需要依赖于操作系统的支持。这使得轻量级线程的创建和切换开销较低。

• 映射到内核级线程：可以利用操作系统的调度算法和资源管理机制，提高线程的并发性和性能。

• 灵活性：用户程序可以根据自己的需求和场景来调度线程。用户程序可以自己实现线程的调度算法，根据需要动态地创建、暂停、恢复和销毁线程，从而提供更灵活的并发控制。

两种方法的优劣

多进程编程模式的优劣

优点

- 充分利用计算资源
- 通过增加CPU就可以容易扩充性能
- 各个进程间独立，稳定性高

缺点

- 逻辑控制复杂，需要和主程序交互
- 通信需要跨进程边界，通信代价高
- 需要谨慎设计数据同步机制，避免死锁和安全性问题
- 每个进程有独立的内存，多进程编程使用的内存更多
- 时间空间开销比较大，限制了并发度的提高

优点

- 开销小，创建和销毁一个新线程花费的时间少，切换线程速度快
- 提高应用程序响应速度
- 相互通信无需调用内核，无需共享内存，共享数据更加高效
- 线程之间的通信占用的系统资源较少
- 可以改善程序结构

缺点

- 每个线程与主程序公用地址空间，受限于2GB地址空间
- 需要处理同步互斥问题，代码设计和调试难度大
- 多线程程序难以debug和维护
- 存在数据冲突的风险
- 一个线程的崩溃可能影响到整个程序的稳定性

扩展阅读

创建进程

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa

打开进程

```
HANDLE OpenProcess(
        [in]    DWORD dwDesiredAccess, //访问安全属性
        [in]    BOOL bInheritHandle, //继承属性
        [in]    DWORD dwProcessId);   //进程pid
```

用来打开一个已存在的进程对象，并返回进程的句柄

https://learn.microsoft.com/zh-cn/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess

终止进程

1. 函数返回
2. 进程自己终止自己

```
VOID ExitProcess(UINT fuExitCode); //退出代码
```

3.终止自身进程或其他进程

```
BOOL TerminateProcess(HANDLE hProcess, //进程句柄
                      UINT fuExitCode); //退出代码
```

https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess
https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminateprocess

线程发展的技术路线：用户态线程与内核态线程

用户态与内核态

- 用户空间（User Space）：应用程序可以使用的内存
- 内核空间（Kernel Space）：只有内核程序可以访问的内存空间
- 用户空间中的代码被限制了只能使用一个局部的内存空间，这些程序在用户态（User Mode） 执行。内核空间中的代码可以访问所有内存，这些程序在内核态（Kernal Mode） 执行。

用户态线程（User Level Thread）

- 在用户空间内运行的线程
- 线程的创建、消息传递、线程调度、保存/恢复上下文都由线程库来完成。内核感知不到多线程的存在。内核继续以进程为调度单位，并且给该进程指定一个执行状态（就绪、运行、阻塞等）。

内核态线程（Kernel-Level Thread）

- 线程管理的所有工作（创建和撤销）由操作系统内核完成
- 可以通过系统调用创造一个内核级线程

线程发展的技术路线：pthread库

POSIX Threads

简称Pthreads，POSIX（Portable Operating System Interface，可移植操作系统接口）是IEEE Computer Society为了提高不同操作系统的兼容性和应用程序的可移植性而制定的一套标准。Pthreads是线程的POSIX标准。定义了一套C程序语言的类型、函数和常量。定义在pthread.h头文件和一个线程库里，大约有100个API，所有API都带有"pthread_"前缀，分为四类：

- **线程管理（Thread management）**：包括线程创建（creating）、分离（detaching）、连接（joining）及设置和查询线程属性的函数等。
- **互斥锁（Mutex）**："mutual exclusion"的缩写，用了限制线程对共享数据的访问，保护共享数据的完整性。包括创建、销毁、锁定和解锁互斥锁及一些用于设置或修改互斥量属性等函数。
- **条件变量（Condition variable）**：用于共享一个互斥量的线程间的通信。包括条件变量的创建、销毁、等待和发送信号（signal）等函数。
- **读写锁（read/write lock）和屏障（barrier）**：包括读写锁和屏障的创建、销毁、等待及相关属性设置等函数。

https://sourceware.org/pthreads-win32/manual/index.html

线程发展的技术路线：C++11之后的thread

C++ 11 Thread

原先C语言中有pthread用来进行多线程编程，但比较偏向于底层。C++ 11 之后有了标准的线程库：std::thread，通过创建一个thread对象来管理C++程序中的多线程

线程创建

- 可通过默认构造函数、接受函数及其传递参数的构造函数和move构造函数创建，一般采用第二种。

```
template <class Fn, class... Args>
explicit thread(Fn&& fn, Args&&... args);
```

- 该 std::thread 对象可被 joinable，新产生的线程会调用Fn 函数，该函数的参数由 args 给出

其他成员函数

- get_id: 获取线程 ID，返回一个类型为 std::thread::id 的对象
- joinable: 检查线程是否可被 join
- detach: 将当前线程对象所代表的执行实例与该线程对象分离，使得线程的执行可以单独进行

https://en.cppreference.com/w/cpp/thread/thread

本章作业

程序设计

利用C语言编程实现对多个文件的扫描

1. 利用fork/exec系统调用实现多进程对文件内容的扫描统计
2. 利用pthread库实现多线程对文件内容的扫描统计

观察并完成实验报告

1. 利用free统计系统多进程和多线程模式的内存消耗
2. 利用top查看系统的资源利用率在不同状态下的变化
3. 利用ipcs实现对共享资源的管理
4. 思考并尝试统计创建一个进程和创建一个线程的时间差

感谢阅读