周哥教 IT.多线程多进程

视频课地址：https://ke.qq.com/course/251408?tuin=a71606

目录

线程的概念-1

进程
所谓线程，即进程中执行运算的最小单位，亦即执行处理机调度的基本单位。如果把进程理解为在逻辑上操作系统所完成的任务，那么线程表示完成该任务的许多可能的子任务之一。线程可以在处理器上独立调度执行，这样，在多处理器环境下就允许几个线程各自在单独处理器上进行。操作系统提供线程就是为了方便而有效地实现这种并发性，线程带来的好处包括：
1）易于调度。线程是系统调度的基本单位，线程的切换比进程要快。
2）提高并发性。通过线程可方便有效地实现并发性。进程可创建多个线程来执行同一程序的不同部分。（左右手画画，唱歌）
3）开销少。创建线程比创建进程要快，所需开销很少。仅占有少量的资源如栈和寄存器。
4）利于充分发挥多处理器的功能。通过创建多线程进程（即一个进程可具有两个或更多个线程），每个线程在一个处理器上运行，从而实现应用程序的并发性，使每个处理器都得到充分运行。

线程的概念-2

那么在引入了线程的系统中，进程和线程的关系是什么样的呢？
1）一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。
2）进程是拥有资源的基本单位，同一进程的所有线程共享该进程的所有资源。
3）线程拥有 CPU，即在 CPU 上运行的是线程。运行一定时间片就会被切换
4）线程在运行过程中，需要同步与互斥。不同进程的线程间要利用进程通信的办法实现同步。
线程作为进程内的一个执行单元，也是进程内的可调度实体。那么它与进程的区别包含哪些

呢？

1）调度与资源分配：线程是系统调度的基本单位，进程是拥有资源的基本单位。线程除了拥有部分寄存器和栈外，不拥有系统资源，但可以访问隶属于进程的资源。

2）系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销。

在系统中，一个程序至少有一个进程，一个进程至少有一个线程。

超线程

超线程是 Intel 所研发的一种技术，于 2002 年发布。超线程的英文是 HT 技术，全名为 Hyper-Threading，中文又名超线程

采用超线程即是可在同一时间里，应用程序可以使用芯片的不同部分。虽然单线程芯片每秒钟能够处理成千上万条指令，但是在任一时刻只能够对一条指令进行操作。而超线程技术可以使芯片同时进行多线程处理，使芯片性能得到提升。

超线程技术是在一颗 CPU 同时执行多个程序而共同分享一颗 CPU 内的资源，理论上要像两颗 CPU 一样在同一时间执行两个线程，P4 处理器需要多加入一个 Logical CPU Pointer（逻辑处理单元）。因此新一代的 P4 HT 的 die 的面积比以往的 P4 增大了 5%。而其余部分如 ALU（算数逻辑运算单元）、FPU（浮点运算单元）、L2 Cache（二级缓存）则保持不变，这些部分是被分享的。

虽然采用超线程技术能同时执行两个线程，但它并不象两个真正的 CPU 那样，每个 CPU 都具有独立的资源。当两个线程都同时需要某一个资源时，其中一个要暂时停止，并让出资源，直到这些资源闲置后才能继续。因此超线程的性能并不等于两颗 CPU 的性能。（连体人例子）

Die ： CPU 核心，又称为内核，就是 CPU 上面中间的小方块，里面是 CPU 的核心，是 CPU 最重要的组成部分。CPU 中心那块隆起的芯片就是核心，是由单晶硅以一定的生产工艺制造出来的，CPU 所有的计算、接受/存储命令、处理数据都由核心执行。各种 CPU 核心都具有固定的逻辑结构，一级缓存、二级缓存、执行单元、指令级单元和总线接口等逻辑单元都会有科学的布局。 为了便于 CPU 设计、生产、销售的管理，CPU 制造商会对各种 CPU 核心给出相应的代号，这也就是所谓的 CPU 核心类型。

核心面积（这是决定 CPU 成本的关键因素，成本与核心面积基本上成正比）

CPU:单核，单核超线程，多核

多线程 3 个重要部分

多线程编程都包含三个组成部分：

一是线程执行函数；（线程代码）

二是线程创建函数；（如何创建线程）

三是线程数据同步与互斥机制 （如何保证多线程数据安全）

其中，线程执行函数就是在新的线程中要执行的代码，需要通过编程去实现，把要在新的线程中完成的任务放在该执行函数中去完成；创建线程函数负责新的线程的创建，它需要线程执行函数做为参数之一，再加上线程执行函数自己的参数也做为参数之一（如果线程执行函

数自己存在参数的话）；线程数据同步机制则负责多线程执行环境条件下数据的完整性和一致性

创建一个线程

```
int g_iValue = 0;
DWORD WINAPI ThreadProc(void* arg)
{
	for (int i = 0; i < 5; i++)
	{
		g_iValue ++;
		printf(" g_iValue = %d\n",
		g_iValue);
	}
	return 1;
}
int main(int argc, char* argv[])
{
	HANDLE hArray[2] = {0};


	hArray[0]=CreateThread(NULL,0,ThreadProc,NULL,0,NULL);
	hArray[1]=CreateThread(NULL,0,ThreadProc,NULL,0,NULL);

	WaitForMultipleObjects(2, hArray, TRUE, INFINITE);

	CloseHandle(hArray[0]);
	CloseHandle(hArray[1]);

	return 0;
}
```

Win32 SDK：查询使用方法-msdn

```
#include <windows.h>
#include <process.h>
```

创建一个 R3 线程（2）

```
#include <process.h>
UINT WINAPI ThreadProc(LPVOID lpParameter)
{
		return 0;
}
```

```
unsigned tid = 0;
HANDLE hThread = (HANDLE)_beginthreadex( NULL, 0, ThreadProc,    NULL, 0, &tid);
WaitForSingleObject( hThread, INFINITE );
//WaitForMultipleObjects(2, hArray, TRUE, INFINITE);
CloseHandle(hThread);
hThread = NULL;
```

给线程传参

创建线程的函数

CreateThread：是 Windows 的 API 函数(SDK 函数的标准形式,直截了当的创建方式，任何场合都可以使用)，提供操作系统级别的创建线程的操作，且仅限于工作者线程。不调用 MFC 和 RTL 的函数时，可以用 CreateThread，其它情况不要使用。因为：
C Runtime 中需要对多线程进行纪录和初始化，以保证 C 函数库工作正常。
MFC 也需要知道新线程的创建，也需要做一些初始化工作。
有些 CRT 的函数象 malloc(),fopen(),_open(),strtok(),ctime(),或 localtime()等函数需要专门的线程局部存储的数据块，这个数据块通常需要在创建线程的时候就建立，如果使用 CreateThread，这个数据块就没有建立，但函数会自己建立一个，然后将其与线程联系在一起，这意味着如果你用 CreateThread 来创建线程，然后使用这样的函数，会有一块内存在不知不觉中创建，而且这些函数并不将其删除，而 CreateThread 和 ExitThread 也无法知道这件事，于是就会有 Memory Leak，在线程频繁启动的软件中，迟早会让系统的内存资源耗尽。
_beginthreadex：MS 对 C Runtime 库的扩展 SDK 函数，首先针对 C Runtime 库做了一些初始化的工作，以保证 C Runtime 库工作正常。然后，调用 CreateThread 真正创建线程。
AfxBeginThread：MFC 中线程创建的 MFC 函数，首先创建了相应的 CWinThread 对象，然后调用 CWinThread::CreateThread,在 CWinThread::CreateThread 中，完成了对线程对象的初始化工作，然后，调用_beginthreadex(AfxBeginThread 相比较更为安全)创建线程。它让线程能够响应消息，可用于界面线程，也可以用于工作者线程。
pthread_create ：LINUX 平台
三个函数的应用条件
AfxBeginThread：在 MFC 中用，工作者线程/界面线程
_beginthreadex: 调用了 C 运行库的，应该用这个，但不能用在 MFC 中。
CreateThread：工作者线程，MFC 中不能用，C Runtime 中不能用。所以任何时候最好都不要用。
AfxBeginThread _beginthreadex CreateThread
线程执行函数长期执行

```
bool g_bwillexit= FALSE;
```

```
UINT WINAPI ThreadProc(void *arg)
{
    while(g_bwillexit==FALSE)
    {
        //music
         //download
    }
}

g_bwillexit=TRUE;
```

多线程安全：家里兄弟姐妹玩一个玩具，打架
多线程安全
带来什么问题？
程序一致性（线程安全性，存钱的例子）
什么样的代码是多线程安全的？（可重入的）
局部变量
全局变量以及资源　加锁（卫生间）
同步？
互斥？（锁）

WINDOWS 同步互斥机制

互斥（竞争）
CRITICAL_SECTION
Mutex
同步（协作）
EVENT
原子操作
InterlockedExchange:g_a=b;
InterlockedIncrement:g_a++;
InterlockedIncrement(i);//i++

CRITICAL_SECTION：临界区

```
CRITICAL_SECTION    cs;
InitializeCriticalSection(&cs);

EnterCriticalSection(&cs);//加锁
//访问全局资源的代码
//只有获得锁的线程可以进入访问
```

```
……
LeaveCriticalSection(&cs);//放锁

DeleteCriticalSection(&cs);
//demo
//i++是否多线程安全？
//死锁
//只能同一个进程中的多个线程之间互斥
A,B 两把锁
线程 1，和线程 2 去拿锁的时候，
都应该遵循先拿 A,再拿 B，先放 B，再放 A
否则，容易造成死锁
int g_iValue=0;


g_iValue++;


mov eax, g_iValue;
add eax,1;
mov g_iValue,eax;

//InterlockedIncrement(&g_iValue);
将 cs 封装为一个 CAutoLocker
class CLock
{
public:
	void Lock() {EnterCriticalSection(&m_sec);}
	void Unlock() {LeaveCriticalSection(&m_sec);}
	CLock () {InitializeCriticalSection(&m_sec);}
	~ CLock () {DeleteCriticalSection(&m_sec);}
private:
	CRITICAL_SECTION m_sec;
};

class CAutoLock
{
public:
	CAutoLock(CLock    * lpLock) :
	m_pLock (lpLock)
	{
```

```
            m_pLock ->Lock();
        }
        ~CAutoLock()
        {
            m_pLock ->Unlock();
        }
private:
    CLock * m_pLock;
};
```

使用例子：

```
CLock mylock;

{
CAutoLock a(&mylock);// 创建了 CAutoLock 的一个对象 a，调用构造函数加锁
    ….
    //当代码结束的时候，对象 a 会销毁，CAutoLock 的析构函数会被调用，解锁
}
```

将单实例改为多线程安全

http://www.mallocfree.com/interview/cpp-20-autolock.htm

http://www.mallocfree.com/interview/design-1-single.htm

15-mtmp\demo\windows\CAutoLock_demo\CAutoLock_Demo

将栈改为多线程安全

Mutex：互斥体

```
HANDLE hMutex = NULL;
hMutex = CreateMutex(NULL, FALSE, NULL);
Name:”Global\\mymutex”

WaitForSingleObject(hMutex, INFINITE);//加锁
//WaitForMultipleObjects(2, hArray, TRUE, INFINITE);

ReleaseMutex(hMutex);//放锁

CloseHandle(hMutex);//关闭
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms686927(v=vs.85).aspx

线程忘记放锁(ReleaseMutex(hMutex))，但是线程结束之后，会自动放锁

可以用于跨进程的线程互斥

EVENT：事件，用作同步

HANDLE g_hEvent;
g_hEvent=CreateEvent(NULL,FALSE,FALSE,NULL);
name:”Global\\myevent”
A:SetEvent(g_hEvent);
B:WaitForSingleObject(g_hEvent, INFINITE);

CloseHandle(g_hEvent);

DEMO：Driver/Seller
原子操作
整数增减赋值：
InterlockedExchange：g_iValue=a;
InterlockedIncrement/InterlockedDecrement
i++/i--这个是非原子操作

R3 多线程同步总结

Critical Section/Mutex/Semaphore/Event
1． Critical Section 与 Mutex 作用非常相似，但 Mutex 是可以命名的，也就是说它可以跨越进程使用。所以创建互斥量需要的资源更多，如果只为了在进程内部使用的话，使用临界区会带来速度上的优势并能够减少资源占用量。因为互斥量是跨进程的，互斥量一旦被创建，就可以通过名字打开它。 Critical Section 只能用在同一个进程的各个线程互斥
2．互斥量（Mutex），信号量（Semaphore），事件（Event）都可以跨越进程来进行同步数据操作（一个进程创建之后，另外的进程可以通过名字打开它，从而用于进程间的数据同步）
3．通过 Mutex 可以指定资源被独占的方式使用，但如果一个资源允许 N（N>1)个进程或者线程访问，这时候如果利用 Mutex 就没有办法完成这个要求， Semaphore 可以，是一种资源计数器。

多进程通信

多个进程需要同步与互斥吗？
多个线程之间通信?
（知识的融会贯通）
多进程通信
有时候，多个进程之间需要进行数据交换，比如 A 进程的一些数据，需要发给 B 进程进行处理。那么数据怎么才能发送给 B 呢？这就是多进程通信的问题。

也就是说：进程与进程在运行期间，也需要数据的交换，即进程通信

进程间的地址空间相对独立。进程与进程间不能像线程间通过全局变量通信。 如果想进程间通信，就需要其他机制。

把数据从外面系统拷贝到虚拟机内部这种数据的交换就叫通信
Windbg+虚拟机调试内核驱动，也就是一种进程通信(管道)

进程间通信方式

共享内存（最快）

管道（pipe,有名，无名）
信号量
消息队列：MFC 中 SendMessage()
文件
端口等
共享内存

```
TCHAR szName[]=TEXT("Global\\MyFileMappingObject");
  hMapFile = CreateFileMapping(szName…);
pBuf = (LPTSTR) MapViewOfFile(hMapFile…)
Read/Write
UnmapViewOfFile(pBuf);
CloseHandle(hMapFile);
```

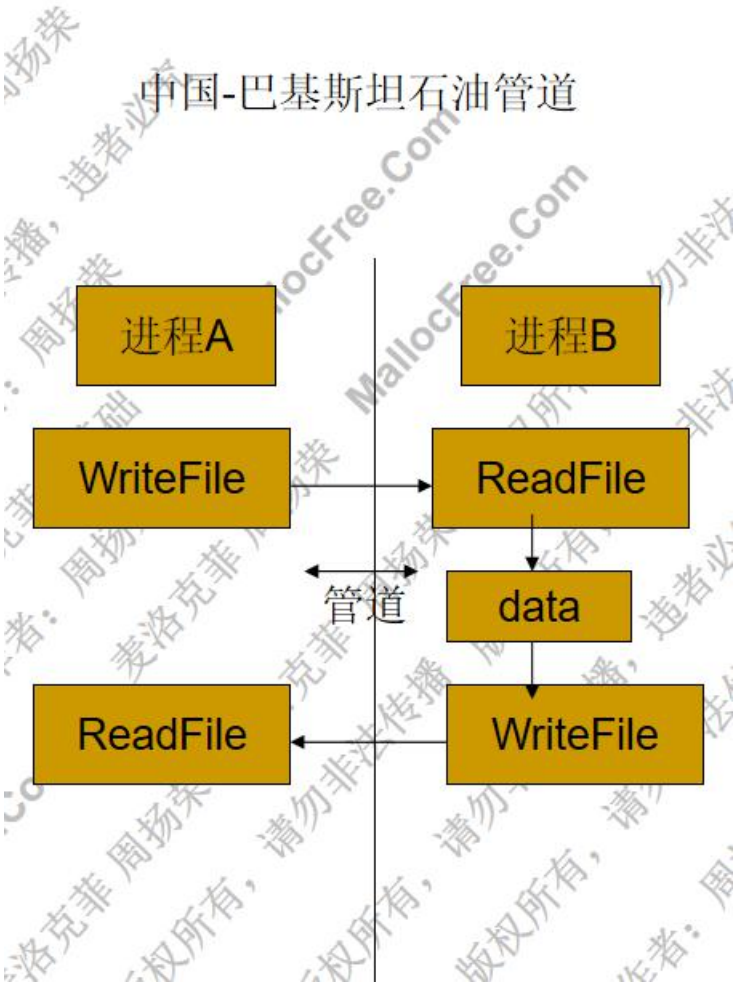
作业：
2 个进程之间通过共享内存传输一个文件（event+mutex）

命名管道

管道是基于文件描述符的通信方式。1 个进程向管道中写的内容被管道另一端的进程读出。1 个进程写入的内容每次都添加在管道缓冲区的末尾，另 1 个进程每次都是从缓冲区的头部读出数据。
全双工

```
#define PIPE_NAME "\\\\.\\Pipe\\test"
CreateNamedPipe()
ConnectNamedPipe()
WriteFile()
```

ReadFile()



匿名管道

管道是基于文件描述符的通信方式。当一个管道建立时，它会创建两个文件描述符 hReadPipe]和 hWritePipe]。其中 hReadPipe]固定用于读管道，而 hWritePipe]固定用于写管道

```
BOOL WINAPI CreatePipe(
_Out_   PHANDLE           hReadPipe,
_Out_   PHANDLE           hWritePipe,
 _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
_In_    DWORD             nSize );
```

ReadFile/WriteFile

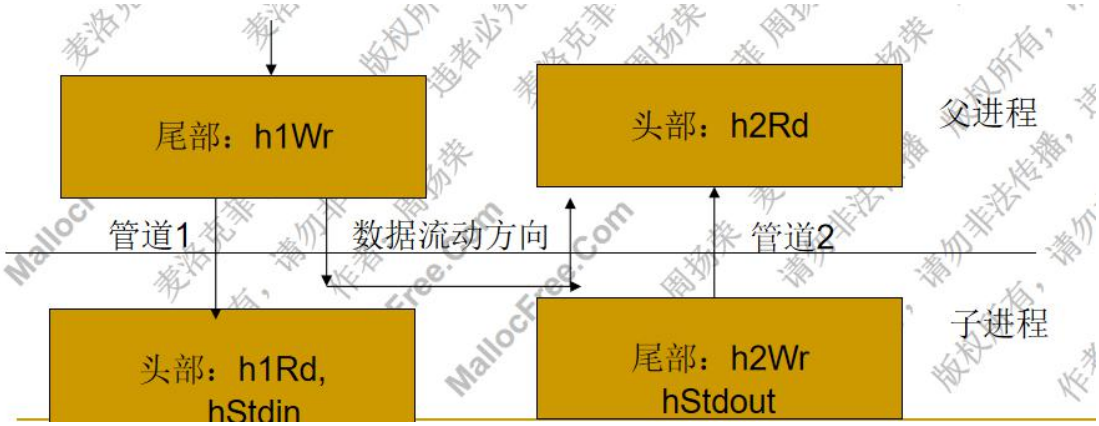
半双工模式

只能用于父子进程或者兄弟进程之间(具有亲缘关系的进程)

父进程打开一个文件读出文件内容，通过管道 1 发送给子进程

子进程通过管道 1 读取数据，然后把数据通过管道 2 发送给父进程

父进程通过管道 2 读出数据并显示



信号量

```
//A：创建
HANDLE semaphore = CreateSemaphore(NULL,
                1,//初始计数，表示可用资源个数
                2, //最大计数，表示最多可用资源个数，如果为 1，互斥体
                _T("Global\\TestSemaphore"));
//B：多个实例打开并使用共享资源
HANDLE semaphore = OpenSemaphore(
      SEMAPHORE_ALL_ACCESS,
      FALSE, _T("Global\\TestSemaphore"));
WaitForSingleObject(semaphore, INFINITE);//对计数减一
//….使用共享资源
ReleaseSemaphore(semaphore, 1, NULL);//使用完对计数加一
//海底捞排队吃火锅的例子
利用信号量启动单实例程序

      m_hSem = CreateSemaphore(NULL, 1, 1, AfxGetApp()->m_pszAppName);      // 信号量已存在？
      // 信号量存在，则程序已有一个实例运行
      if (GetLastError() == ERROR_ALREADY_EXISTS)
      {
            // 关闭信号量句柄
            CloseHandle(m_hSem);
            m_hSem = NULL;
            //MessageBox(NULL, L"程序已经运行", L"Error", MB_OK);
            HWND hWnd = ::FindWindow(NULL, _T("PopupClient"));
                  if (hWnd)
```

```
            {
                ::SetForegroundWindow(hWnd);
                ::ShowWindow(hWnd, SW_SHOW);
            }
            // 前一实例已存在，但找不到其主窗
            // 可能出错了
            // 退出本实例
            return FALSE;
    }
```

//安装程序需要，安全软件

作业

C++的构造函数和析构函数实现一个自动互斥锁

2 个进程之间通过共享内存传输一个文件（event+lock）

将 singleton 设计模式改为多线程安全

参考 http://cantellow.iteye.com/blog/838473

LINUX 中创建一个线程

```
#include <stdio.h>
#include <pthread.h>

void *thread_function(void *dummyPtr)
{
    printf("Thread number %ld\n", pthread_self());
}

int main(int argc, char *argv[])
{
    pthread_t thread_id;
    pthread_create( &thread_id, NULL, thread_function, NULL );

    pthread_join( thread_id, NULL);

    printf("Final\n");
    return 0;
}
```

Linux 多线程编程

pthread_create()/pthread_join/pthread_detach
创建一个线程默认的状态是 joinable，如果一个线程结束运行但没有被 join,则它的状态类似于进程中的 Zombie Process,即还有一部分资源没有被回收（退出状态码），所以创建线程者应该调用 pthread_join 来等待线程运行结束，并可得到线程的退出代码，回收其资源（类似于 wait,waitpid)
但是调用 pthread_join(pthread_id)后，如果该线程没有运行结束，调用者会被阻塞，在有些情况下我们并不希望如此，比如在 Web 服务器中当主线程为每个新来的链接创建一个子线程进行处理的时候，主线程并不希望因为调用 pthread_join 而阻塞（因为还要继续处理之后到来的链接），这时可以在子线程中加入代码 pthread_detach(pthread_self())或者父线程调用 pthread_detach(thread_id)（非阻塞，可立即返回）
多线程互斥

```
pthread_mutex_t mutex;      //定义锁，保护 g_counter
int g_counter = 0; //全局变量 g_counter
pthread_mutex_init(&mutex, NULL);//初始化锁

pthread_mutex_lock( &mutex );//拿锁
g_counter++;
pthread_mutex_unlock( &mutex )//放锁
```

线程同步

```
pthread_mutex_t count_lock;
pthread_cond_t count_nz_event;
unsigned count = 0;
void * decrement_count    (void *dummyPtr) {
      pthread_mutex_lock (&count_lock);
      while(count==0) {
            //count_lock 在该函数内部有一个放锁和加锁的过程,否则 increment 线程无法进入
            pthread_cond_wait( &count_nz_event, &count_lock);
       }
       count=count -1;
       pthread_mutex_unlock (&count_lock);
}
void * increment_count(void *dummyPtr){
      pthread_mutex_lock(&count_lock);
      if(count==0)
            pthread_cond_signal(& count_nz_event);
      count=count+1;
      pthread_mutex_unlock(&count_lock);
}
int main(int argc, char* argv[])
```

```
{
        pthread_mutex_init(&count_lock, NULL);
        pthread_cond_init(& count_nz_event, NULL);
        pthread_t p1, p2;
        pthread_create(&p1, NULL, decrement_count, NULL);
        pthread_create(&p2, NULL, increment_count, NULL);
}
```

LINUX 进程通信

线程怎么通信的？
进程通信
有时候，多个进程之间需要进行数据交换，比如 A 进程的一些数据，需要发给 B 进程进行处理。那么数据怎么才能发送给 B 呢？这就是多进程通信的问题。

也就是说：进程与进程在运行期间，也需要数据的交换，即进程通信

进程间的地址空间相对独立。进程与进程间不能像线程间通过全局变量通信。 如果想进程间通信，就需要其他机制。

把数据从外面系统拷贝到虚拟机内部这种数据的交换就叫通信
Windbg+虚拟机调试内核驱动，也就是一种进程通信
共享内存
管道
信号量

LINUX 进程通信-共享内存

共享内存就是允许两个不相关的进程访问同一个逻辑内存。共享内存是在两个正在运行的进程之间共享和传递数据的一种非常有效的方式。不同进程之间共享的内存通常安排为同一段物理内存。进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用 C 语言函数 malloc 分配的内存一样。而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。
但是，共享内存并未提供同步机制，也就是说，在第一个进程结束对共享内存的写操作之前，并无自动机制可以阻止第二个进程开始对它进行读取(通过信号量）
共享内存是进程通信最快的方式
Mmap:适合于父子进程:demo\mmapshare\b //a is wrong
Posix： ftok（创建一个共享内存的 key） shmget（根据 key 拿到 id） shmat（根据 id 拿到共享内存的地址） shmdt（释放 detach 这个共享内存的地址） 父子进程或者不相关进程

LINUX 进程通信-管道

管道是基于文件描述符的通信方式。1 个进程向管道中写的内容被管道另一端的进程读出。写入的内容每次都添加在管道缓冲区的末尾，并且每次都是从缓冲区的头部读出数据。
管道是基于文件描述符的通信方式。当一个匿名管道通过 pipe()建立时，它会创建两个文件描述符 fd[0]和 fd[1]。其中 fd[0]固定用于读管道，而 fd[1]固定用于写管道
mkfifo open：有名管道
Pipe：无名管道
匿名管道只能用于父子进程或者兄弟进程之间(具有亲缘关系的进程)
LINUX 进程通信-信号量
信号量:一个资源允许 N（N>1)个进程或者线程访问，这时候 Semaphore 可以是一种资源计数器。
ftok（创建一个共享内存的 key）
semget //通过 key 创建 SEM 集
semctl //初始化或者获取 SEM 值
semop//P(-1)/V(+1)操作
信号量模拟 mutex

无锁化编程技术

从一道阿里面试题说起

无锁化编程有哪些常见方法？（）
A.针对计数器，可以使用原子加，比如 AtomicInteger __sync_fetch_and_add
B.只有一个生产者和一个消费者，那么就可以做到免锁访问环形缓冲区（Ring Buffer）
C.RCU（Read-Copy-Update），新旧副本切换机制，对于旧副本可以采用延迟释放的做法
D.CAS（Compare-and-Swap），如无锁栈，无锁队列等待
虽然是一道选择题，却引出了无锁化编程的大部分内容，不了解这个技术的，估计不知所云。
多线程开发加锁基本原理
多线程带来什么问题？
程序一致性（线程安全性，存钱的例子）
什么样的代码是多线程安全的？（可重入的）
局部变量
全局变量以及资源　加锁（卫生间）
互斥？（锁）

临界区和互斥体

```
CRITICAL_SECTION    cs;
InitializeCriticalSection(&cs);
int g_counter = 0;
EnterCriticalSection(&cs);//加锁
//访问全局资源的代码
//只有获得锁的线程可以进入访问
g_counter++;
LeaveCriticalSection(&cs);//放锁

DeleteCriticalSection(&cs);
HANDLE hMutex = NULL;
hMutex = CreateMutex(NULL, FALSE, NULL);
Name:”Global\\mymutex”

WaitForSingleObject(hMutex, INFINITE);//加锁
//访问全局资源的代码
g_counter++;
ReleaseMutex(hMutex);//放锁

CloseHandle(hMutex);//关闭
```

https://msdn.microsoft.com/en-us/library/windows/desktop/ms686927(v=vs.85).aspx
线程忘记放锁(ReleaseMutex(hMutex))，但是线程结束之后，会自动放锁
可以用于跨进程的线程互斥

Linux 中多线程互斥

```
pthread_mutex_t mutex;      //定义锁，保护 g_counter
int g_counter = 0; //全局变量 g_counter
pthread_mutex_init(&mutex, NULL);//初始化锁

pthread_mutex_lock( &mutex );//拿锁
g_counter++;
pthread_mutex_unlock( &mutex )//放锁
```

Java 多线程互斥：synchronized

1，代码段互斥：

Object lock=new Object();
synchronized (lock) {…}
2，函数互斥
public synchronized void func(){…}
3，对变量互斥
final Person p =new Person();
synchronized (p) {…}

悲观锁与乐观锁

悲观锁：总做最坏的打算（悲观），每次访问数据时都认为别人会修改，所以每次在拿数据的时候都会上锁，这样其他人想访问这个数据就会阻塞直到获得该锁为止。关系型数据库里的行锁，表锁等，读锁，写锁等，都是悲观锁，即在做操作之前先上锁。Java 里面的同步原语 synchronized 和 ReentrantLock 的实现也是悲观锁。
select * from mytable for update
public synchronized void func(){…}
乐观锁：每次访问数据的时候都认为别人不会修改（乐观），所以不上锁，在更新时需要判断此期间有没有别人更新这个数据，比如使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高效率。乐观锁的缺点是不能解决脏读的问题。
CAS：java.util.concurrent.atomic 包下面的原子变量类
update mytable set Name='tom',version=version+1 where ID=#{id} and version=#{version}
如果并发量不大且不允许脏读，可以使用悲观锁解决并发问题。但如果系统的并发非常大的话，悲观锁定会带来非常大的性能问题，这个时候我们就要选择乐观锁。
加锁带来的问题

死锁

两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。
死锁不能自行解开

活锁

进程（或线程）在执行过程中没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。
比如某些重试机制导致一个交易（请求）被不断地重试，而每次重试都是失败的（线程在做无用功）
小狗追着自己的尾巴咬，虽然一直在咬却一直没有咬到；两个人在窄路相遇，同时向一个方向避让，然后又向另一个方向避让，如此反复。

活锁有可能自行解开

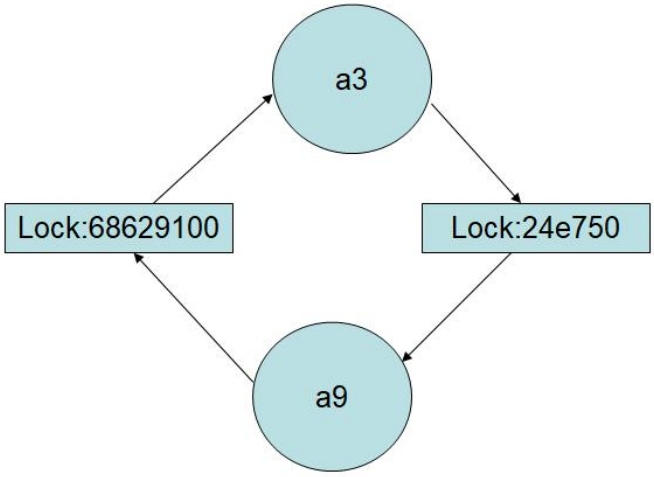
饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。如果事务 T1 封锁了数据 R,事务 T2 又请求封锁 R，T2 等待。T3 也请求封锁 R，当 T1 释放了 R 上的封锁后，系统首先批准了 T3 的请求，T2 仍然等待。然后 T4 又请求封锁 R，当 T3 释放了 R 上的封锁之后，系统又批准了 T4 的请求.....T2 可能永远等待。（先来先服务）

优先级翻转:T2 如果优先级高，就会造成优先级翻转，造成高优先级任务被许多具有较低优先级任务阻塞，实时性难以得到保证。解决方法：优先级天花板；优先级继承等。

性能下降

拿锁放锁耗时

拿锁期间无法并发，阻塞了其他线程



i++	3.2亿
lock(p_mutex);i++;unlock(p_mutex);	2千万
CAS_atomic_add1(i)	4千万
interlockedIncrease(&i)	4千万

什么是无锁化编程？

无锁化编程(Lock-Free)：不使用锁的情况下实现多线程之间对变量进行同步和访问的一种程序设计实现方法，Lock-Free 的程序不包括锁机制，但不包括锁机制的程序不一定是 lock-free 的。

同步阻塞编程（Block）

基于锁的（lock-based）

同步非阻塞编程（Non-blocking Synchronization）：

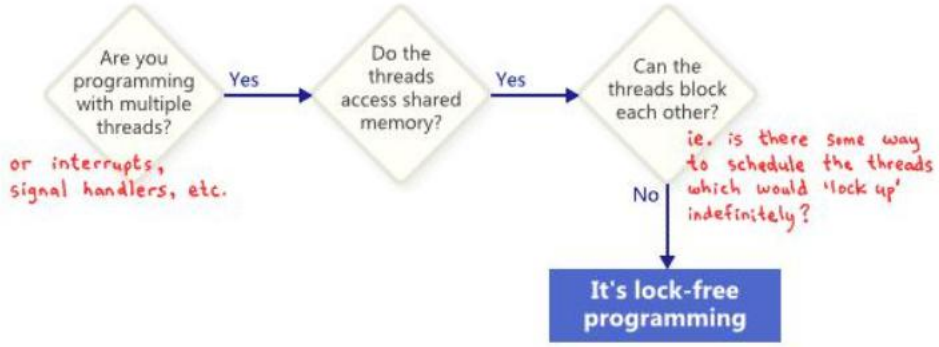
Wait-free: 所有线程可以在有限步之内结束

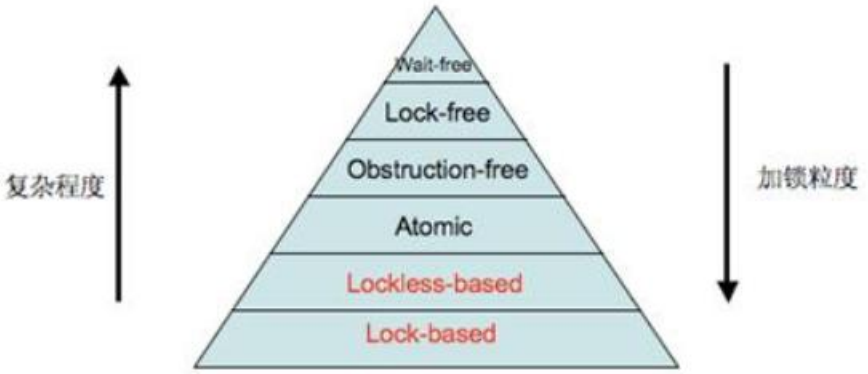
Lock-free：任何时刻至少有一个线程可以在有限步内完成操作。

Obstruction-free：一个线程，在其它线程都暂停（无竞争）的情况下，可以在有限步之内结束。

所有 wait free 都是 lock free，所有的 lock free 都是 obstruction free

异步编程





无锁编程涉及的技术要点

无锁编程具体实现的时候涉及到的技术包括：原子操作（atomic operations），内存屏障（memory barriers），内存顺序（memory order）， 指令序列一致性（sequential consistency）和 ABA 现象处理

无锁编程

原子操作

CAS

Ring buffer

RCU

无锁编程之原子操作

非原子操作

i++;

读、改、写（回）

movl x, %eax

addl $1, %eax

movl %eax, x

i=0；

原子操作：在执行完毕之前不会被任何其它任务或事件中断的一系列操作。它是非阻塞编程的基础，没有原子操作，会因为中断异常等各种原因引起数据状态的不一致从而影响到程序的正确。

硬件层面原子操作：

基本的内存操作的原子性：

CPU 保证处理器从内存当中读取或者写入一个字节的行为肯定是原子的

其他 CPU 处理器不能访问这个字节的内存地址

复杂的内存操作 CPU 不能保证其原子性，比如跨总线宽度或者跨多个缓存行（Cache Line），跨页表的访问

CPU 指令集中的原子操作指令保证：如 X86 平台下中的是 CMPXCHG，就是一种 CAS 原子操作，属于 Read-Modify-Write （RMW）类型

操作系统层面原子操作（基于硬件层面的原子操作）：

LINUX:atomic_set/and/inc

Windows:InterlockedExchange/InterlockedIncrement

各种开发语言中（c,c++,java）基于操作系统提供的接口

原子操作的顺序性及问题

原子操作不仅要保证操作的原子性，还要考虑在不同的语言和内存模型下如何保证操作的顺序性， 编译时和运行时的指令重排序，内存顺序冲突（Memory order violation）等问题。

原子性确保指令执行期间不被打断，要么全部执行，要么根本不执行

顺序性确保即使两条或多条指令出现在独立的执行线程中或者独立的处理器上时，保持它们本该执行的顺序

伪代码：

x=y=0;

线程 1：

x=1;

r1=y;

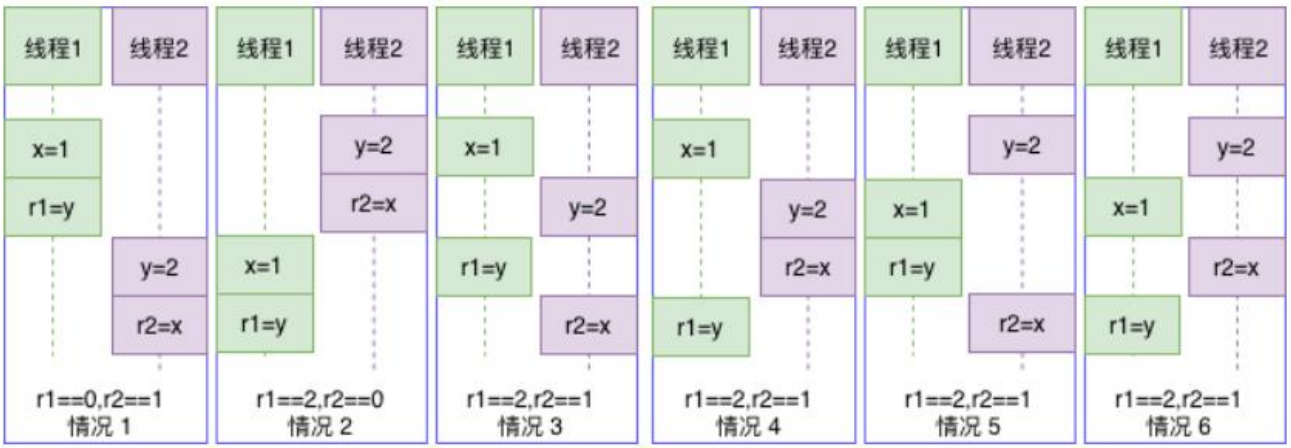
线程 2：

y=2;

r2=x;

在编译器、CPU 不对指令进行重排，且两个线程交织执行(假设以上四条语句都是原子操作)时共有 4!/(2!*2!)=6 种情况

再考虑乱序执行的情况下：Core1 的指令预处理单元看到线程 1 的两条语句没有依赖性(不管哪条语句先执行，在两条指令语句完成后都会得到一样的结果)，会先执行 r1=y 再执行 x=1，或者两条指令同时执行。Core2 也一样。这样一来就有可能出来 r1==r2==0 的结果。



原子操作顺序性问题解决方法

加锁，串行执行，但效率不高，优先级反转等问题

原子操作，C++11 内存模型

synchronized-with

假设 X 是一个原子变量。如果线程 A 写了 X，线程 B 读了 X，那么线程 A、B 间存在 synchronized-with 关系，它能保证对 X 的读和写是互斥的。

happens-before 机制

i=0；

i = 1;　　//线程 A 执行

j = i ;　　//线程 B 执行

j 是否一定等于 1 呢？假定线程 A 的操作（i = 1）happens-before 线程 B 的操作（j = i）,那么可以确定线程 B 执行后即使在乱序等情况下 j = 1 一定成立

C++11std:atomic 对不同的内存 order 对原子性和 synchronized-with 和 happens-before 机制有不同的定义 https://www.jianshu.com/p/7d237771dc94

auto r1 = y.load(std::memory_order_relaxed);

x.store(r1, std::memory_order_relaxed);

序号	内存模型	memory_order值	备注
1	宽松	memory_order_relaxed	只要求原子操作，不需要其它同步保障
2	释放-获取	memory_order_acquire memory_order_release memory_order_acq_rel	
3	释放-消费	memory_order_consume	C++20起
4	顺序一致	memory_order_seq_cst	默认内存序

无锁化编程之 CAS 操作

CAS(Compare and Swap，即比较并替换），wikipedia 中对于 CAS 的定义为：

1、是一种多线程中的原子操作
2、类似于乐观锁，只有当读取的值和预期值相等时才进行赋新值
3、返回结果必须表明是否成功

CAS 操作包含三个操作数：内存地址（V）、预期原值（A）、新值（B）。如果内存地址的值与预期原值相同，那么处理器会自动将内存的值更新为新值。否则，处理器不做任何操作。无论哪种情况，处理器都会在 CAS 指令之前返回该地址的值。CAS 有效地说明了“我认为地址 V 应该包含值 A；如果包含该值，则将 B 放到这个地址；否则，不要更新该地址，只告诉我这个地址现在的值即可。”

代码：

```
bool compare_and_swap(int *reg, int oldval, int newval){
    int reg_val = *reg;
    if(reg_val == oldval) {
        *reg = newval;
        return true;
    }
    return false;
}
```

几乎所有的 CPU 指令都支持 CAS 的原子操作，X86 下对应的是 CMPXCHG 汇编指令

适用场景：

CAS 适合简单对象的操作，比如布尔值、整型值等；

CAS 适合冲突较少的情况，如果太多线程在同时自旋，那么长时间循环会导致 CPU 开销很大；

CAS 在各个平台的支持

1）Linux 的 CAS

GCC4.1+版本中支持 CAS 的原子操作

```
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
```

2）Windows 的 CAS

```
InterlockedCompareExchange ( __inout LONG volatile *Target,
                             __in LONG Exchange,
                             __in LONG Comperand);
```

3) C++11 中的 CAS

C++11 中的 STL 中的 atomic 类的函数可以跨平台

```
template class <T>
bool atomic_compare_exchange_weak( std::atomic <T>* obj,
```

```
                                        T* expected, T desired );
    template class <T>
    bool atomic_compare_exchange_weak( volatile std::atomic <T>* obj,
                                        T* expected, T desired );
```

Java 基于 CAS 的应用（1）：Atomic

Java 并发包中许多 Atomic 的类的底层原理都是 CAS。在 java.util.concurrent 包中大量实现都是建立在基于 CAS 实现 Lock-Free 算法上，没有 CAS 就不会有此包。Java.util.concurrent.atomic 提供了基于 CAS 实现的若干原语

```
AtomicInteger atomicInteger = new AtomicInteger(5);
atomicInteger.compareAndSet(5, 10)；
```

getAndAddInt， incrementAndGet 在底层 Unsafe 类中的代码，运用到了 CAS

```
public final int getAndAddInt(Object obj, long obj_addr, int val) {
        int old;
        do {
            old = this.getIntVolatile(obj, obj_addr);
        } while(!this.compareAndSwapInt(obj, obj_addr, old, old + val));
                return old;
            }
  public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Java 基于 CAS 的应用（2）：自旋锁

自旋锁也用到了 CAS

```
public class SpinLock{
    private AtomicReference<Thread> owner =new AtomicReference<>();
    private int count =0;
    public void lock(){
        Thread current = Thread.currentThread();
        if(current==owner.get()){
            count++;
            return ;
```

```
            }
            while(!owner.compareAndSet(null, current)){

            }
        }
        public void unlock (){
            Thread current = Thread.currentThread();
            if(current==owner.get()){
                if(count!=0){
                    count--;
                }else{
                    owner.compareAndSet(current, null);
                }
            }
        }
}
```

Java 多线程中 synchronized 锁升级

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

锁的升级的目的：锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

在 JavaSE 1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。synchronized 的底层实现主要依靠 Lock-Free 的队列，基本思路是自旋后阻塞，竞争切换后继续竞争锁。在线程冲突较少的情况下，可以获得和 CAS 类似的性能；而线程冲突严重的情况下，性能远高于 CAS。

CAS 面临的问题

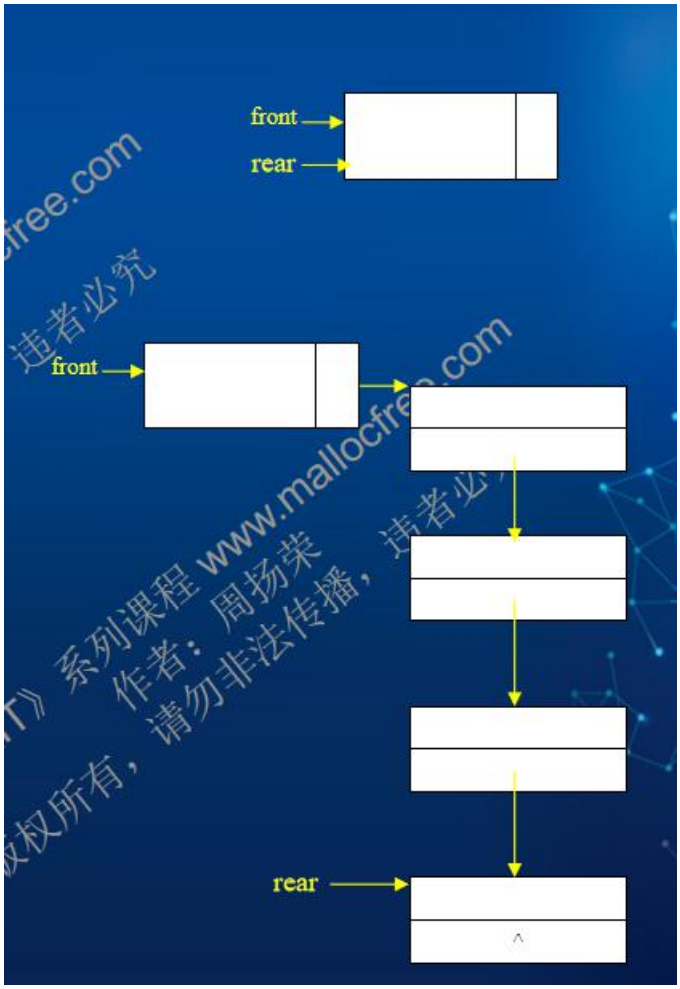
1，ABA 问题：如果内存地址 V 初次读取的值是 A，在 CAS 等待期间它的值曾经被改成了 B，后来又被改回为 A，那 CAS 操作就会误认为它从来没有被改变过（手提箱例子）。ABA 问题以及解决：使用带版本号的原子引用 AtomicStampedRefence<V>

```
AtomicReference<String> atomicReference = new AtomicReference<>("A");
AtomicStampedReference<String> stampReference = new AtomicStampedReference<>("A",1);

线程 1：
int stamp = stampReference.getStamp();//1

atomicReference.compareAndSet("A","B");
atomicReference.compareAndSet("B","A");

stampReference.compareAndSet("A","B",stamp,stamp+1);
stampReference.compareAndSet("B","A",stamp+1,stamp+2);
stampReference.getStamp();//3

线程 2：
int stamp = stampReference.getStamp();//1
atomicReference.compareAndSet("A","C");
atomicReference.get();//C

stampReference.compareAndSet("A","C",stamp,stamp+1);//fail, 1!=3
stampReference.getReference();//A
2，高并发下，反复更新某个变量，却不成功，导致 CPU 负载过高
3，只保证对一个变量进行原子操作，多个变量无能为力，只能加锁了。
```

CAS 实现无锁队列

```
int cas_deque(LinkQue *q, int *e){
    QNode *tmp=NULL;
    do {
        if (is_que_empty(q)){
            return(0);
        }

        tmp = q->front->next;
    } while (!CAS((long *)(&(q->front->next)), (long)tmp, (long)tmp->next));

    *e = tmp->data;
    free(tmp);
    return(1);
}

void cas_enque(LinkQue *q, int e){
```

```
    QNode *newNode = (QNode *)malloc(sizeof(QNode));
    newNode->data = e;
    newNode->next = NULL;

    QNode *tmp;
    do{
        tmp = q->rear;
    }while (!CAS((long *)(&(tmp->next)), NULL, (long)newNode));
    q->rear = newNode;
}
```



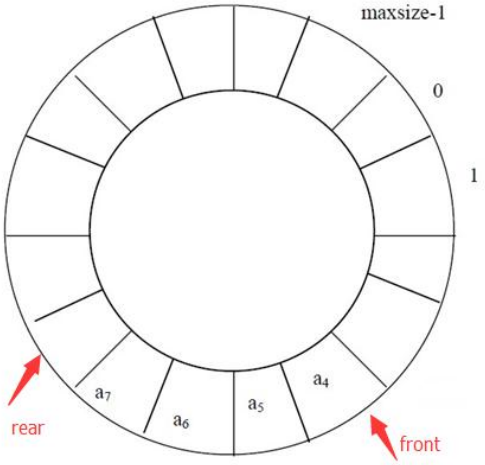
无锁队列 Demo

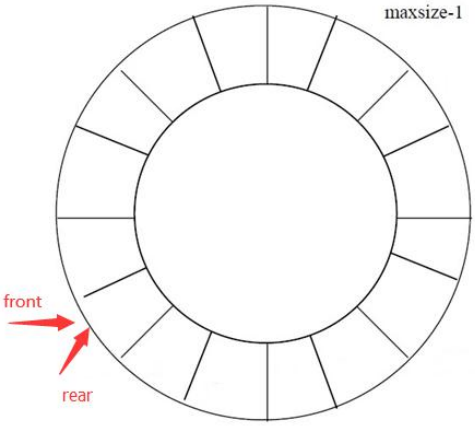
无锁编程之环形缓冲区（ ring buffer ）
如果只有一个生产者和一个消费者，那么就可以做到免锁访问环形缓冲区（Ring Buffer）。生产者将数据放入数组的尾端，而消费者从数组的另一端（头部）移走数据，当达到数组的尾部时，生产者绕回到数组的头部。

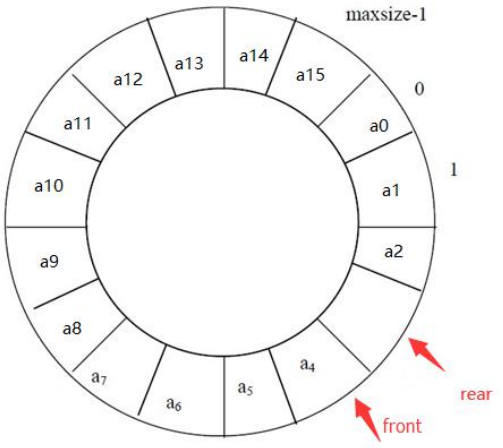
写入索引（或指针，rear）只允许生产者访问并修改，只要写入者在更新索引之前将新的值保存到缓冲区中，则读者将始终看到一致的数据。同理，读取索引（front）也只允许消费者访问并修改。

基于数组的循环队列

```
int EnQueue(int value);//rear = ( rear+1 ) % MAXSIZE;
int DeQueue(int *value);//front = (front+1) % MAXSIZE;
int IsQueueFull();//(rear+1) % MAXSIZE == front
int    IsQueueEmpty();//rear==front
```







Linux kfifo

在 Linux 内核文件 kfifo.h 和 kfifo.c 中，定义了一个先进先出环形缓冲区实现 kfifo。

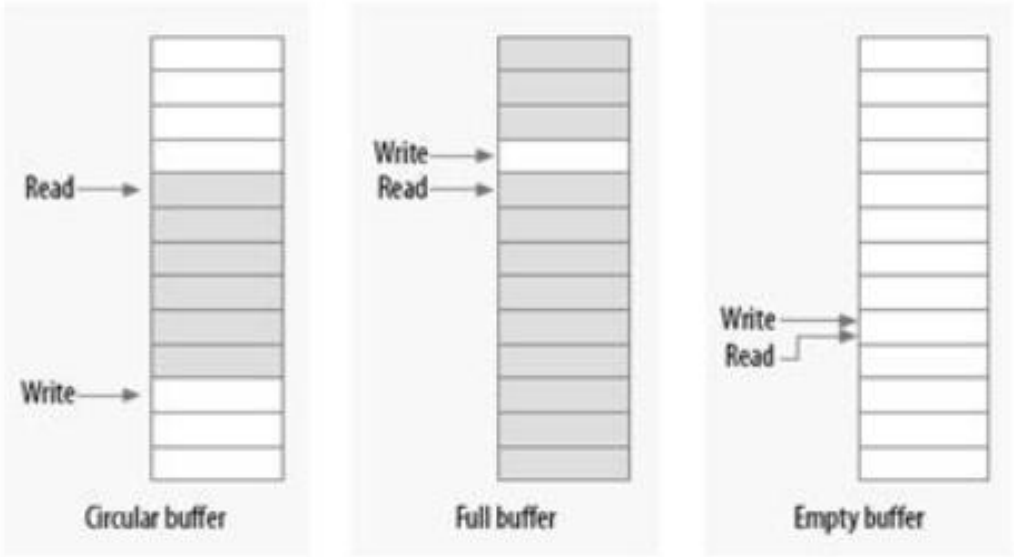
kfifo 缓冲区长度为 2 的幂。读、写指针分别是无符号整型变量。把读写指针变换为缓冲区内的索引值：指针值 & （缓冲区长度-1）。比“求余”操作高效。读指针和写指针地址关系：

读指针 + 缓冲区存储的数据长度 == 写指针

kfifo 的写操作时计算缓冲区剩余可写空间长度：

len = min{待写入数据长度，缓冲区长度 - （写指针 - 读指针）}

然后分两段写入数据：第一段是从写指针开始向缓冲区末尾方向；第二段是从缓冲区起始处写入余下的可写入数据，这部分可能数据长度为 0 即并无实际数据写入。



无锁编程之 RCU

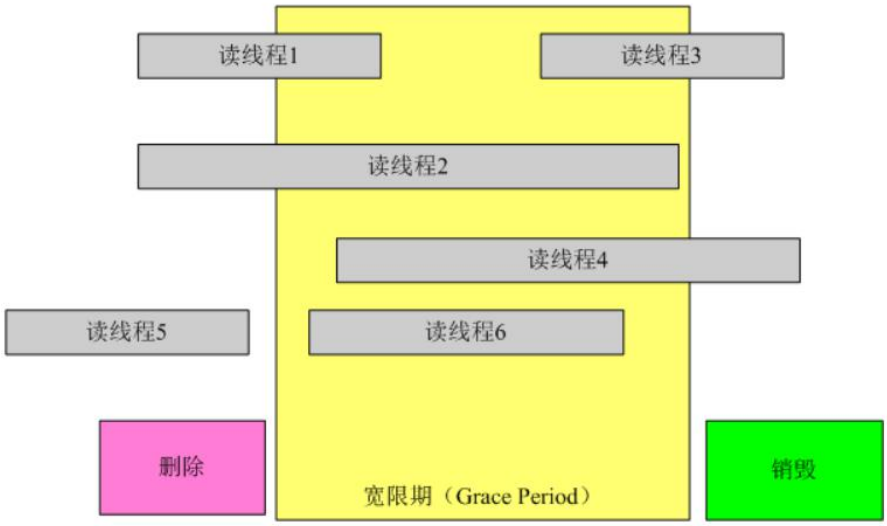
RCU（Read-Copy Update）主要针对链表（堆上分配的内存），读取数据时不对链表进行加锁，允许多个线程同时读取，而只能一个线程对链表进行修改（加锁），在 Linux 内核中有广泛

应用
适用于读多写少，比如 FS 的目录
RCU 的实现需要考虑：
宽限期：在读取一个节点过程中，另外一个线程删除了这个节点。删除线程可以把这个节点从链表中移除(remove)，但不能立即销毁它(free)，必须等到所有的读取线程完成后。这个过程称为宽限期（Grace period）。
节点完整性：如果读线程读到了另一线程插入的一个新节点，需要保证读线程读到的这个节点的完整性。
链表完整性：新增或者删除一个节点，不会导致遍历一个链表从中间断开。但并不保证一定能读到新增的节点或者不读到要被删除的节点

无锁编程之 RCU（1）：宽限期

```
node *g_data;
DEFINE_SPINLOCK(mutex);
void work_read(void){
	node *fp = g_data;
	if( fp != NULL ){//此处切换
		dosth(fp->a, fp->b, fp->c);
	}
}
void work_update(node * new_fp){
	spin_lock(&mutex);
	node *old_fp = g_data;
	g_data = new_fp;
	spin_unlock(&mutex);
}
void work_read(void){
	rcu_read_lock(); //帮助检查宽限期是否结束
	node *fp = g_data;
	if( fp != NULL ){//此处切换
		dosth(fp->a, fp->b, fp->c);
	}
    rcu_read_unlock();
}
void work_update(node * new_fp){
	spin_lock(&mutex);
	node *old_fp = g_data;
	g_data = new_fp;
	spin_unlock(&mutex);
	synchronize_rcu();  //宽限期开始，等待所有线程 rcu_read_unlock();
```

```
    kfree(old_fp);
}
```



线程 5 宽限期前就读结束了；线程 3，4，6 宽限期后才开始读；线程 1，2 在宽限期开始前开始读，宽限期开始时尚未结束

无锁编程之 RCU（2）：节点完整性问题

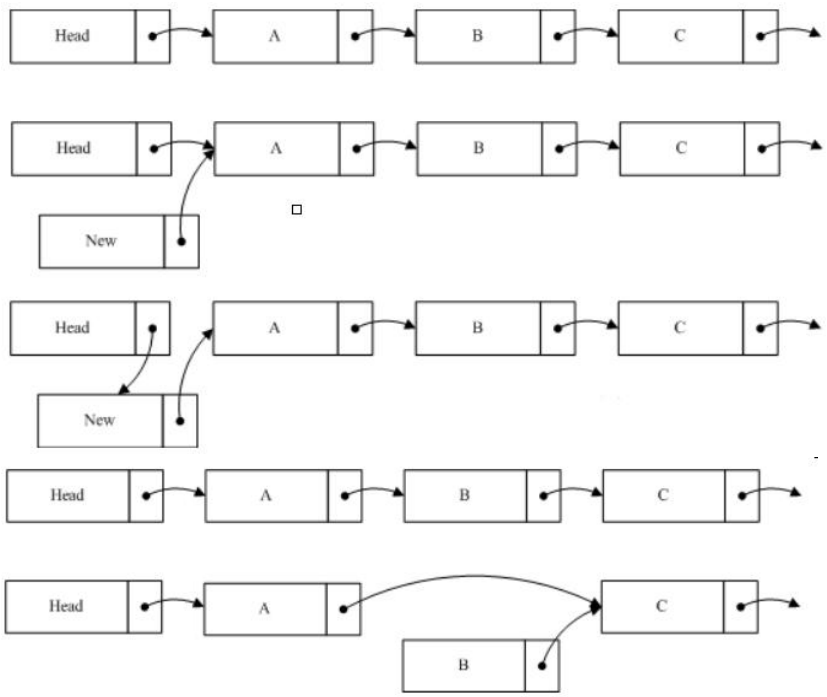
```
void work_read(void){
    rcu_read_lock(); //帮助检查宽限期是否结束
    node *fp = g_data;
    if( fp != NULL ){//先于 rcu_read_lock()执行
        dosth(fp->a, fp->b, fp->c);
    }
   rcu_read_unlock();
}
void work_update(node * new_fp){//乱序执行
    spin_lock(&mutex);
    node *old_fp = g_data;
    new_fp->a=…
    new_fp->b=…
    new_fp->c=…
    g_data = new_fp;
    spin_unlock(&mutex);
    synchronize_rcu();  //宽限期开始，等待所有线程 rcu_read_unlock();
    kfree(old_fp);
}
```

问题 1：work_update 乱序执行，优化屏障

rcu_assign_pointer(g_data,new_fp);保证执行顺序，发布
问题 2： work_read(void)乱序执行，导致部分数据 old 和 new
node *fp = rcu_dereference(g_data); 替换 node *fp = g_data;，订阅

无锁编程之 RCU（3）：链表完整性

插入节点：先 new->next=A;再 Head->next=new;
删除节点：先 A->next=C;保持 B->next=C;宽限期保证读 B 的线程可以继续访问后续节点。



无锁编程与有锁编程性能对比

在性能上基于 CAS 实现的硬件级的互斥，其单次操作性能比相同条件下的应用层的较为高效，但当多个线程并发时，硬件级的互斥引入的消耗一样很高(类似 spin_lock)。 无锁算法及相关数据结构并不意味在所有的环境下都能带来整体性能的极大提升。循环 CAS 操作对时会大量占用 cpu，对系统时间的开销也是很大。这也是基于循环 CAS 实现的各种自旋锁不适合做操作和等待时间太长的并发操作的原因。而通过对有锁程序进行合理的设计和优化，在很多的场景下更容易使程序实现高度的并发性。
只有少数数据结构可以支持实现无锁编程，比如队列、栈、链表、词典等
通过对锁之间的数据进行批处理，可以极大的提高系统的性能，而使用原子操作，则无法实现批处理上的改进
//加锁与批处理
lock();
for(k=0;k<100;k++)

```
    i++;
unlock();

//CAS 原子操作
for(k=0;k<100;k++)
    InterlockedIncrement(i);
```

当多个线程并发时，硬件级的互斥引入的代价与应用层的锁争用同样令人惋惜。因此如果纯粹希望通过使用 CAS 无锁算法及相关数据结构而带来程序性能的大量提升是不可能的，硬件级原子操作使应用层操作变慢，而且无法再度优化。相反通过对有锁多线程程序的良好设计，可以使程序性能没有任何下降，可以实现高度的并发性。

但是我们也要看到应用层无锁的好处，比如不需要程序员再去考虑死锁、优先级反转等问题，因此在对应用程序不太复杂，而对性能要求稍高时，可以采用有锁多线程。而程序较为复杂，性能要求满足使用的情况下，可以使用应用级无锁算法

无锁算法常见开源库

自己开发，还是使用开源库？大厂还是小厂

MidiShare Source Code is available under the GPL license. MidiShare includes implementations of lock-free FIFO queues and LIFO stacks.

Appcore is an SMP and HyperThread friendly library which uses Lock-free techniques to implement stacks, queues, linked lists and other useful data structures. Appcore appears currently to be for x86 computers running Windows. The licensing terms of Appcore are extremely unclear.

Noble – a library of non-blocking synchronisation protocols. Implements lock-free stack, queue, singly linked list, snapshots and registers. Noble is distributed under a license which only permits non-commercial academic use.（commercial version）

lock-free-lib published under the GPL license. Includes implementations of software transactional memory, multi-workd CAS primitives, skip lists, binary search trees, and red-black trees. For Alpha, Mips, ia64, x86, PPC, and Sparc.

Nonblocking multiprocessor/multithread algorithms in C++ (for MSVC/x86) posted by Joshua Scholar to musicdsp.org , and are presumably in the public domain. Included are queue, stack, reference-counted garbage collection, memory allocation, templates for atomic algorithms and types. This code is largely untested. A local mirror is here .

Qprof includes the Atomic_ops library of atomic operations and data structures under an MIT-style license. Only available for Linux at the moment, but there are plans to support other platforms. download available here

Amino Concurrent Building Blocks provides lock free datastructures and STM for C++ and Java under an Apache Software (2.0) licence.

http://www.lmax.com/disruptor

《The Art of Multiprocessor Programming》（多处理器编程的艺术）（美）Maurice Herlihy / Nir Shavit