**linux惊群效应**

详细的介绍什么是惊群，惊群在线程和进程中的具体表现，惊群的系统消耗和惊群的处理方法。

**1、惊群效应是什么？**

       惊群效应也有人叫做雷鸣群体效应，不过叫什么，简言之，惊群现象就是多进程（多线程）在同时阻塞等待同一个事件的时候（休眠状态），如果等待的这个事件发生，那么他就会唤醒等待的所有进程（或者线程），但是最终却只可能有一个进程（线程）获得这个时间的“控制权”，对该事件进行处理，而其他进程（线程）获取“控制权”失败，只能重新进入休眠状态，这种现象和性能浪费就叫做惊群。

        为了更好的理解何为惊群，举一个很简单的例子，当你往一群鸽子中间扔一粒谷子，所有的各自都被惊动前来抢夺这粒食物，但是最终注定只可能有一个鸽子满意的抢到食物，没有抢到的鸽子只好回去继续睡觉，等待下一粒谷子的到来。这里鸽子表示进程（线程），那粒谷子就是等待处理的事件。

**看一下：**[*WIKI的雷鸣群体效应的解释*](https://en.m.wikipedia.org/wiki/Thundering_herd_problem)

**2.惊群效应到底消耗了什么？**

     我想你应该也会有跟我一样的问题，那就是惊群效应到底消耗了什么？

     （1）、系统对用户进程/线程频繁地做无效的调度，上下文切换系统性能大打折扣。

     （2）、为了确保只有一个线程得到资源，用户必须对资源操作进行加锁保护，进一步加大了系统开销。

     是不是还是觉得不够深入，概念化？看下面：

         \*1、上下文切换（context  switch）过高会导致cpu像个搬运工，频繁地在寄存器和运行队列之间奔波，更多的时间花在了进程（线程）切换，而不是在真正工作的进程（线程）上面。直接的消耗包括cpu寄存器要保存和加载（例如程序计数器）、系统调度器的代码需要执行。间接的消耗在于多核cache之间的共享数据。

**看一下：**[*wiki上下文切换*](https://en.m.wikipedia.org/wiki/Context_switch)

         \*2、通过锁机制解决惊群效应是一种方法，在任意时刻只让一个进程（线程）处理等待的事件。但是锁机制也会造成cpu等资源的消耗和性能损耗。目前一些常见的服务器软件有的是通过锁机制解决的，比如nginx（它的锁机制是默认开启的，可以关闭）；还有些认为惊群对系统性能影响不大，没有去处理，比如lighttpd。

**3.惊群效应的庐山真面目。**

**让我们从进程和线程两个方面来揭开惊群效应的庐山真面目：**

**\*1）accept（）惊群：**

       首先让我们先来考虑一个场景：

        主进程创建了socket、bind、listen之后，fork()出来多个进程，每个子进程都开始循环处理（accept）这个listen\_fd。每个进程都阻塞在accept上，当一个新的连接到来时候，所有的进程都会被唤醒，但是其中只有一个进程会接受成功，其余皆失败，重新休眠。

       那么这个问题真的存在吗？

       历史上，Linux的accpet确实存在惊群问题，但现在的内核都解决该问题了。即，当多个进程/线程都阻塞在对同一个socket的接受调用上时，当有一个新的连接到来，内核只会唤醒一个进程，其他进程保持休眠，压根就不会被唤醒。

       不妨写个程序测试一下，眼见为实：

fork\_thunder\_herd.c:

**[cpp]** [view plain](http://blog.csdn.net/lyztyycode/article/details/78648798) [copy](http://blog.csdn.net/lyztyycode/article/details/78648798)

1. #include<stdio.h>
2. #include<stdlib.h>
3. #include<sys/types.h>
4. #include<sys/socket.h>
5. #include<sys/wait.h>
6. #include<string.h>
7. #include<netinet/in.h>
8. #include<unistd.h>
10. #define PROCESS\_NUM 10
11. int main()
12. {
13. int fd = socket(PF\_INET, SOCK\_STREAM, 0);
14. int connfd;
15. int pid;
17. char sendbuff[1024];
18. **struct sockaddr\_in serveraddr;**
19. serveraddr.sin\_family = AF\_INET;
20. serveraddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);
21. serveraddr.sin\_port = htons(1234);
22. bind(fd, (**struct sockaddr \*)&serveraddr, sizeof(serveraddr));**
23. listen(fd, 1024);
24. int i;
25. **for(i = 0; i < PROCESS\_NUM; ++i){**
26. pid = fork();
27. **if(pid == 0){**
28. **while(1){**
29. connfd = accept(fd, (**struct sockaddr \*)NULL, NULL);**
30. snprintf(sendbuff, **sizeof(sendbuff), "接收到accept事件的进程PID = %d\n", getpid());**
32. send(connfd, sendbuff, strlen(sendbuff)+1, 0);
33. printf("process %d accept success\n", getpid());
34. close(connfd);
35. }
36. }
37. }
38. //int status;
39. wait(0);
40. **return 0;**
41. }

这个程序模拟上面的场景，当我们用telnet连接该服务器程序时，会看到只返回一个进程pid，即只有一个进程被唤醒。

我们用strace -f来追踪fork子进程的执行：

编译：cc fork\_thunder\_herd.c -o server

           一个终端执行strace -f  ./server  你会看到如下结果（只截取部分可以说明问题的截图，减小篇幅）：

这里我们首先看到系统创建了十个进程。下面这张图你会看出十个进程阻塞在accept这个系统调用上面：

接下来在另一个终端执行telnet 127.0.0.1 1234：

很明显当telnet连接的时候只有一个进程accept成功,你会不会和我有同样的疑问，就是会不会内核中唤醒了所有的进程只是没有获取到资源失败了，就好像惊群被“隐藏”？

这个问题很好证明，我们修改一下代码：

**[cpp]** [view plain](http://blog.csdn.net/lyztyycode/article/details/78648798) [copy](http://blog.csdn.net/lyztyycode/article/details/78648798)

1. connfd = accept(fd, (**struct sockaddr \*)NULL, NULL);**
2. **if(connfd == 0){**
4. snprintf(sendbuff, **sizeof(sendbuff), "接收到accept事件的进程PID = %d\n", getpid());**
6. send(connfd, sendbuff, strlen(sendbuff)+1, 0);
7. printf("process %d accept success\n", getpid());
8. close(connfd);
9. }**else{**
10. printf("process %d accept a connection failed: %s\n", getpid(), strerror(errno));
11. close(connfd);
12. }

没错，就是增加了一个accept失败的返回信息，按照上面的步骤运行，这里我就不截图了，我只告诉你运行结果与上面的运行结果无异，增加的失败信息并没有输出，也就说明了这里并没有发生惊群，所以注意阻塞和惊群的唤醒的区别。

**Google了一下：**其实在linux2.6版本以后，linux内核已经解决了accept（）函数的“惊群”现象，大概的处理方式就是，当内核接收到一个客户连接后，只会唤醒等待队列上的第一个进程（线程）,所以如果服务器采用accept阻塞调用方式，在最新的linux系统中已经没有“惊群效应”了

accept函数的惊群解决了，下面来让我们看看存在惊群现象的另一种情况：epoll惊群

**\*2）epoll惊群：**

概述：

如果多个进程/线程阻塞在监听同一个监听socket fd的epoll\_wait上，当有一个新的连接到来时，所有的进程都会被唤醒。  
同样让我们假设一个场景：  
主进程创建socket，bind，listen后，将该socket加入到epoll中，然后fork出多个子进程，每个进程都阻塞在epoll\_wait上，如果有事件到来，则判断该事件是否是该socket上的事件如果是，说明有新的连接到来了，则进行接受操作。为了简化处理，忽略后续的读写以及对接受返回的新的套接字的处理，直接断开连接。  
那么，当新的连接到来时，是否每个阻塞在epoll\_wait上的进程都会被唤醒呢？  
很多博客中提到，测试表明虽然epoll\_wait不会像接受那样只唤醒一个进程/线程，但也不会把所有的进程/线程都唤醒。

这究竟是问什么呢？

**看一下：**[**多进程epoll和“惊群”**](http://blog.163.com/pandalove@126/blog/static/9800324520122633515612)

我们还是眼见为实，一步步解决上面的疑问：

代码实例：epoll\_thunder\_herd.c:

**[cpp]** [view plain](http://blog.csdn.net/lyztyycode/article/details/78648798) [copy](http://blog.csdn.net/lyztyycode/article/details/78648798)

1. #include<stdio.h>
2. #include<sys/types.h>
3. #include<sys/socket.h>
4. #include<unistd.h>
5. #include<sys/epoll.h>
6. #include<netdb.h>
7. #include<stdlib.h>
8. #include<fcntl.h>
9. #include<sys/wait.h>
10. #include<errno.h>
11. #define PROCESS\_NUM 10
12. #define MAXEVENTS 64
13. //socket创建和绑定
14. int sock\_creat\_bind(char \* port){
15. int sock\_fd = socket(AF\_INET, SOCK\_STREAM, 0);
16. **struct sockaddr\_in serveraddr;**
17. serveraddr.sin\_family = AF\_INET;
18. serveraddr.sin\_port = htons(atoi(port));
19. serveraddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);
21. bind(sock\_fd, (**struct sockaddr \*)&serveraddr, sizeof(serveraddr));**
22. **return sock\_fd;**
23. }
24. //利用fcntl设置文件或者函数调用的状态标志
25. int make\_nonblocking(int fd){
26. int val = fcntl(fd, F\_GETFL);
27. val |= O\_NONBLOCK;
28. **if(fcntl(fd, F\_SETFL, val) < 0){**
29. perror("fcntl set");
30. **return -1;**
31. }
32. **return 0;**
33. }
35. int main(int argc, char \*argv[])
36. {
37. int sock\_fd, epoll\_fd;
38. **struct epoll\_event event;**
39. **struct epoll\_event \*events;**
41. **if(argc < 2){**
42. printf("usage: [port] %s", argv[1]);
43. exit(1);
44. }
45. **if((sock\_fd = sock\_creat\_bind(argv[1])) < 0){**
46. perror("socket and bind");
47. exit(1);
48. }
49. **if(make\_nonblocking(sock\_fd) < 0){**
50. perror("make non blocking");
51. exit(1);
52. }
53. **if(listen(sock\_fd, SOMAXCONN) < 0){**
54. perror("listen");
55. exit(1);
56. }
57. **if((epoll\_fd = epoll\_create(MAXEVENTS))< 0){**
58. perror("epoll\_create");
59. exit(1);
60. }
61. event.data.fd = sock\_fd;
62. event.events = EPOLLIN;
63. **if(epoll\_ctl(epoll\_fd, EPOLL\_CTL\_ADD, sock\_fd, &event) < 0){**
64. perror("epoll\_ctl");
65. exit(1);
66. }
67. /\*buffer where events are returned\*/
68. events = calloc(MAXEVENTS, **sizeof(event));**
69. int i;
70. **for(i = 0; i < PROCESS\_NUM; ++i){**
71. int pid = fork();
72. **if(pid == 0){**
73. **while(1){**
74. int num, j;
75. num = epoll\_wait(epoll\_fd, events, MAXEVENTS, -1);
76. printf("process %d returnt from epoll\_wait\n", getpid());
77. sleep(2);
78. **for(i = 0; i < num; ++i){**
79. **if((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP) || (!(events[i].events & EPOLLIN))){**
80. fprintf(stderr, "epoll error\n");
81. close(events[i].data.fd);
82. **continue;**
83. }**else if(sock\_fd == events[i].data.fd){**
84. //收到关于监听套接字的通知，意味着一盒或者多个传入连接
85. **struct sockaddr in\_addr;**
86. socklen\_t in\_len = **sizeof(in\_addr);**
87. **if(accept(sock\_fd, &in\_addr, &in\_len) < 0){**
88. printf("process %d accept failed!\n", getpid());
89. }**else{**
90. printf("process %d accept successful!\n", getpid());
91. }
92. }
93. }
94. }
95. }
96. }
97. wait(0);
98. free(events);
99. close(sock\_fd);
100. **return 0;**
101. }

上面的代码编译gcc epoll\_thunder\_herd.c -o server

一个终端运行代码 ./server 1234  另一个终端telnet 127.0.0.1 1234

运行结果：

这里我们看到只有一个进程返回了，似乎并没有惊群效应，让我们用strace -f  ./server 8888追踪执行过程（这里只给出telnet之后的截图，之前的截图参考accept，不同的就是进程阻塞在epoll\_wait）

截图（部分）：

运行结果显示了部分个进程被唤醒了，返回了“process accept failed”只是后面因为某些原因失败了。所以这里貌似存在部分“惊群”。

怎么判断发生了惊群呢？

我们根据strace的返回信息可以确定:

1)系统只会让一个进程真正的接受这个连接，而剩余的进程会获得一个EAGAIN信号。图中有体现。

2）通过返回结果和进程执行的系统调用判断。

这究竟是什么原因导致的呢？

看我们的代码，看似部分进程被唤醒了，而事实上其余进程没有被唤醒的原因是因为某个进程已经处理完这个事件，无需唤醒其他进程，你可以在epoll获知这个事件的时候sleep（2）；这样所有的进程都会被唤起。看下面改正后的代码结果更加清晰：

代码修改：

**[cpp]** [view plain](http://blog.csdn.net/lyztyycode/article/details/78648798) [copy](http://blog.csdn.net/lyztyycode/article/details/78648798)

1. num = epoll\_wait(epoll\_fd, events, MAXEVENTS, -1);
2. printf("process %d returnt from epoll\_wait\n", getpid());
3. sleep(2);

运行结果：

如图所示：所有的进程都被唤醒了。所以epoll\_wait的惊群确实存在。

为什么内核处理了accept的惊群，却不处理epoll\_wait的惊群呢？

我想，应该是这样的：  
accept确实应该只能被一个进程调用成功，内核很清楚这一点。但epoll不一样，他监听的文件描述符，除了可能后续被accept调用外，还有可能是其他网络IO事件的，而其他IO事件是否只能由一个进程处理，是不一定的，内核不能保证这一点，这是一个由用户决定的事情，例如可能一个文件会由多个进程来读写。所以，对epoll的惊群，内核则不予处理。

**\*3）线程惊群：**

    进程的惊群已经介绍的很详细了，这里我就举一个线程惊群的简单例子，我就截取上次红包代码中的代码片段，如下

**[cpp]** [view plain](http://blog.csdn.net/lyztyycode/article/details/78648798) [copy](http://blog.csdn.net/lyztyycode/article/details/78648798)

1. printf("初始的红包情况：<个数：%d  金额：%d.%02d>\n",item.number, item.total/100, item.total%100);
2. pthread\_cond\_broadcast(&temp.cond);//红包包好后唤醒所有线程抢红包
3. pthread\_mutex\_unlock(&temp.mutex);//解锁
4. sleep(1);

没错你可能已经注意到了，pthread\_cond\_broadcast()在资源准备好以后，或者你再编写程序的时候设置的某个事件满足时它会唤醒队列上的所有线程去处理这个事件，但是只有一个线程会真正的获得事件的“控制权”。

解决方法之一就是加锁。下面我们来看一看解决或者避免惊群都有哪些方法？

**4.我们怎么解决“惊群”呢？你有什么高见？**

这里通常代码加锁的处理机制我就不详述了，来看一下常见软件的处理机制和linux最新的避免和解决的办法

**（1）、Nginx的解决：**

如上所述，如果采用epoll，则仍然存在该问题，nginx就是这种场景的一个典型，我们接下来看看其具体的处理方法。

nginx的每个worker进程都会在函数ngx\_process\_events\_and\_timers()中处理不同的事件，然后通过ngx\_process\_events()封装了不同的事件处理机制，在Linux上默认采用epoll\_wait()。

在主要ngx\_process\_events\_and\_timers()函数中解决惊群现象。

**[cpp]** [view plain](http://blog.csdn.net/lyztyycode/article/details/78648798) [copy](http://blog.csdn.net/lyztyycode/article/details/78648798)

1. **void ngx\_process\_events\_and\_timers(ngx\_cycle\_t \*cycle)**
2. {
3. ... ...
4. // 是否通过对accept加锁来解决惊群问题，需要工作线程数>1且配置文件打开accetp\_mutex
5. **if (ngx\_use\_accept\_mutex) {**
6. // 超过配置文件中最大连接数的7/8时，该值大于0，此时满负荷不会再处理新连接，简单负载均衡
7. **if (ngx\_accept\_disabled > 0) {**
8. ngx\_accept\_disabled--;
9. } **else {**
10. // 多个worker仅有一个可以得到这把锁。获取锁不会阻塞过程，而是立刻返回，获取成功的话
11. // ngx\_accept\_mutex\_held被置为1。拿到锁意味着监听句柄被放到本进程的epoll中了，如果
12. // 没有拿到锁，则监听句柄会被从epoll中取出。
13. **if (ngx\_trylock\_accept\_mutex(cycle) == NGX\_ERROR) {**
14. **return;**
15. }
16. **if (ngx\_accept\_mutex\_held) {**
17. // 此时意味着ngx\_process\_events()函数中，任何事件都将延后处理，会把accept事件放到
18. // ngx\_posted\_accept\_events链表中，epollin|epollout事件都放到ngx\_posted\_events链表中
19. flags |= NGX\_POST\_EVENTS;
20. } **else {**
21. // 拿不到锁，也就不会处理监听的句柄，这个timer实际是传给epoll\_wait的超时时间，修改
22. // 为最大ngx\_accept\_mutex\_delay意味着epoll\_wait更短的超时返回，以免新连接长时间没有得到处理
23. **if (timer == NGX\_TIMER\_INFINITE || timer > ngx\_accept\_mutex\_delay) {**
24. timer = ngx\_accept\_mutex\_delay;
25. }
26. }
27. }
28. }
29. ... ...
30. (**void) ngx\_process\_events(cycle, timer, flags);   // 实际调用ngx\_epoll\_process\_events函数开始处理**
31. ... ...
32. **if (ngx\_posted\_accept\_events) { //如果ngx\_posted\_accept\_events链表有数据，就开始accept建立新连接**
33. ngx\_event\_process\_posted(cycle, &ngx\_posted\_accept\_events);
34. }
36. **if (ngx\_accept\_mutex\_held) { //释放锁后再处理下面的EPOLLIN EPOLLOUT请求**
37. ngx\_shmtx\_unlock(&ngx\_accept\_mutex);
38. }
40. **if (delta) {**
41. ngx\_event\_expire\_timers();
42. }
44. ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0, "posted events %p", ngx\_posted\_events);
45. // 然后再处理正常的数据读写请求。因为这些请求耗时久，所以在ngx\_process\_events里NGX\_POST\_EVENTS标
46. // 志将事件都放入ngx\_posted\_events链表中，延迟到锁释放了再处理。
47. }}

具体的解释参考：[nginx处理惊群详解](http://m.blog.csdn.net/russell_tao/article/details/7204260)

**（2）、SO\_REUSEPORT**

Linux内核的3.9版本带来了SO\_REUSEPORT特性，该特性支持多个进程或者线程绑定到同一端口，提高服务器程序的性能，允许多个套接字bind()以及listen()同一个TCP或UDP端口，并且在内核层面实现负载均衡。

在未开启SO\_REUSEPORT的时候，由一个监听socket将新接收的连接请求交给各个工作者处理，看图示：

在使用SO\_REUSEPORT后，多个进程可以同时监听同一个IP：端口，然后由内核决定将新链接发送给哪个进程，显然会降低每个工人接收新链接时锁竞争

**下面让我们好好比较一下多进程（线程）服务器编程传统方法和使用SO\_REUSEPORT的区别**

运行在Linux系统上的网络应用程序，为了利用多核的优势，一般使用以下典型的多进程（多线程）服务器模型：

**1.单线程listener/accept，多个工作线程接受任务分发，虽然CPU工作负载不再成为问题，但是仍然存在问题：**

       （1）、单线程listener（图一），在处理高速率海量连接的时候，一样会成为瓶颈

        （2）、cpu缓存行丢失套接字结构现象严重。

**2.所有工作线程都accept（）在同一个服务器套接字上呢？一样存在问题：**

        （1）、多线程访问server socket锁竞争严重。

        （2）、高负载情况下，线程之间的处理不均衡，有时高达3：1。

        （3）、导致cpu缓存行跳跃（cache line bouncing）。

        （4）、在繁忙cpu上存在较大延迟。

上面两种方法共同点就是很难做到cpu之间的负载均衡，随着核数的提升，性能并没有提升。甚至服务器的吞吐量CPS（Connection Per Second）会随着核数的增加呈下降趋势。

下面我们就来看看SO\_REUSEPORT解决了什么问题：

        （1）、允许多个套接字bind()/listen()同一个tcp/udp端口。每一个线程拥有自己的服务器套接字，在服务器套接字上没有锁的竞争。

        （2）、内核层面实现负载均衡

        （3）、安全层面，监听同一个端口的套接字只能位于同一个用户下面。

        （4）、处理新建连接时，查找listener的时候，能够支持在监听相同IP和端口的多个sock之间均衡选择。

当一个连接到来的时候，系统到底是怎么决定那个套接字来处理它？

对于不同内核，存在两种模式，这两种模式并不共存，一种叫做热备份模式，另一种叫做负载均衡模式，3.9内核以后，全部改为负载均衡模式。

**热备份模式：**一般而言，会将所有的reuseport同一个IP地址/端口的套接字挂在一个链表上，取第一个即可，工作的只有一个，其他的作为备份存在，如果该套接字挂了，它会被从链表删除，然后第二个便会成为第一个。  
**负载均衡模式：**和热备份模式一样，所有reuseport同一个IP地址/端口的套接字会挂在一个链表上，你也可以认为是一个数组，这样会更加方便，当有连接到来时，用数据包的源IP/源端口作为一个HASH函数的输入，将结果对reuseport套接字数量取模，得到一个索引，该索引指示的数组位置对应的套接字便是工作套接字。这样就可以达到负载均衡的目的，从而降低某个服务的压力。

编程关于SO\_REUSEPORT的详细介绍请参考：

[SO\_REUSEPORT](http://blog.csdn.net/lyztyycode/article/details/%E7%BB%99%E4%BD%A0%E6%89%BE%E4%BA%86%E4%B8%80%E4%B8%AA%E9%93%BE%E6%8E%A5%EF%BC%8C%E7%9C%8B%E4%B8%8B)

参考资料：

<https://pureage.info/2015/12/22/thundering-herd.html>

<http://www.tuicool.com/articles/2aumqe>

<http://blog.163.com/leyni@126/blog/static/16223010220122611523786/>

<http://baike.baidu.com/link?url=6x0zTazmBxTYE9ngPt_boKjS8ivdQnRlfhHj-STCnqG9tjKwfCluPsKlq-ASUkdQTPW3XrD8FtyilBaI75GJCK>

<http://m.blog.csdn.net/tuantuanls/article/details/41205739>

[tcp对so\_reuseport的优化](http://blog.csdn.net/lyztyycode/article/details/@%E6%9C%9B)