Jerry Xing
A01354731

POSIX 线程

- POSIX 线程在 pthread.h 中，每个线程都有线程ID（TID），类型为（pthread_t），调用 pthread_self 找到TID，pthread_equal 判断相同
- pthread_create 创建的线程可以显式终止：pthread_exit，如果函数或 main 返回将隐式终止，等价于调用前面的 exit：void pthread_exit(void *retval);
- 一个线程可以通过 pthread_join 等待另一个线程终止：int pthread_join(pthread_t tid, void **thread_return);
  - 必须是可 join 的线程，不能是 detached 的
  - 可 join 的线程终止时内存资源不会释放，直到另一个线程调用 pthread_join
  - 用 pthread_detach 将一个线程分离，则线程终止时资源可以立即回收：int pthread_detach(pthread_t tid);

Mutexes 互斥量

- 用于保护 critical section 临界区以便同时只有一个线程执行该代码
  - 执行前锁住互斥量，执行后解锁。
  - 同一时间只有一个线程可以锁，因此实现互斥
  - 互斥类型为 pthread_mutex_t，基本操作有 lock, trylock, unlock
  - 初始化：int mutex = PTHREAD_MUTEX_INITIALIZER
- 注意
  - 不应该复制互斥量的副本，结果未定义
  - 只有拥有互斥量的线程可以解锁，错误的尝试可能返回错误或意外成功
  - 一个线程如果需要同时有多个互斥量要避免死锁：
    - 死锁：多个线程互相等对方释放资源，四个条件全满足才会发生
      - Mutual Exclusion 互斥：资源一次只能由一个线程占用
      - Hold and Wait 占有并等待：线程占用资源后还要等待别的资源
      - No Preemption 不可抢占：线程已经占用的资源不能被抢走，只能等待进程自愿释放
      - Circular Wait 循环等待：线程的等待形成一个封闭环
    - 发生死锁后，程序不会崩溃或报错，但线程也不运行，导致系统卡住
    - Fixed locking hierarchy 固定锁层级策略：所有线程在获取多个锁时，必须按相同顺序获取
    - Try & backoff strategy 尝试并退避策略：锁住第一个互斥量后，使用 pthread_mutex_trylock 锁住其它互斥量，如果失败则解锁所有已获得的互斥量并重新尝试（尝试失败后不是再无限等待，而是赶紧释放自己有的锁避免其它线程在那死锁等待）

Condition Variables 条件变量

- 类型为 pthread_cond_t，与互斥量 Associate 关联，基本操作包括 signal（条件为真时通知条件上的一个线程），wait（等待另一个线程的 signal 时挂起）和 broadcast（通知条件上的全部线程）
- 使用方法

a. 检查条件前先锁住互斥量（防止另一个线程检查后更改数据导致不同步）
b. 如果条件为真，执行任务并释放互斥量，否则 wait 条件并释放互斥量，将线程放到条件变量上睡眠
c. 当条件被 signal 时，一个睡眠线程被唤醒。如果是 broadcast 广播，会唤醒该条件上的所有线程
d. 唤醒的线程从 wait 返回并重新锁住互斥量，检查条件然后重新进行 b.

```
// 等待条件时
pthread_mutex_lock(&mutex);         // 测试前必须获得互斥量
while (condition != TRUE)           // 唤醒后需要重新测试条件
   pthread_cond_wait(&cond, &mutex);
do_thing();
pthread_mutex_unlock(&mutex);
// 发送信号时
pthread_mutex_lock(&mutex);         // 变更条件前必须获得互斥量
condition = TRUE;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond);
```

- wait 必须在 while 循环内调用：当 wait 返回后，虽然有可能是 signal 触发，但也有可能是下面两种，所以要重新检查条件。
  - Spurious Wakeup 虚假唤醒：即使没有 signal，线程也可能被唤醒
  - 条件在被唤醒时已经不成立（Race Condition 竞态条件）
  - 多个线程同时等待时，唤醒的线程不一定拿到资源

Sockets 套接字

- 为网络协议提供编程接口。TCP的数据以段和数据报形式发送，套接字是通信的端点

```
int socket(int domain, int type, int protocol); // 成功时返回引用已创建套接字的描述符，出错时返回-1并设置errno，protocol通常为0
```

  - domain：通信域和协议族，AF/PF_UNIX 用于 Unix 域套接字，AF/PF_INET 用于 Internet 域套接字
  - type：通信语义
    - SOCK_STREAM 基于字节流的有序、可靠的双向链接
    - SOCK_DGRAM 指定对数据报的无连接，不可靠，固定最大长度消息的支持

- API

```
// 创建一个 socket，返回文件描述符 FD
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
// 把 socket 绑定到一个特定的 IP + port，通常只在服务器使用
bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));
// 把一个已绑定的 socket 设置为被动模式，开始监听连接请求。没有它就不能 accept
listen(sockfd, 5);
// accept 会阻塞，直到有客户端来连，返回一个新的 FD
int clientfd = accept(sockfd, NULL, NULL);
// 客户端像服务器发起连接请求，三次握手后可以read/write，服务器没开则阻塞或失败
connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));
// 在已连接的 socket 上读取数据，阻塞直到收到数据(返回读取字节数)或对端（已连接的 socket）关闭(返回0)
read(sockfd, buffer, 1024);
// 发送数据给已连接的 socket，阻塞直到发送完成
write(sockfd, data, len);
// 关闭 socket 释放 FD。服务器一般先关闭单个连接（clientfd），最后关闭 sockfd
close(sockfd);
```

Blocking behaviour 阻塞行为

阻塞意味着一个函数会卡住等待条件满足才返回。Socket中最常见的阻塞点有：

- accept() 阻塞：调用accept后会一直等到有客户端连上来
- connect() 阻塞：客户端尝试连接服务器时，如果服务器没开或网络不通，则会阻塞直到超时
- read() 阻塞：会卡住直到对方发送数据或关闭连接（return 0）

常见 **bug**

- 重复绑定（bind: Address already in use）

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));
bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));  // 重复绑定
```

- 文件描述符泄漏（File Descriptor Leak）

打开一个 socket（也就是一个文件描述符fd）但没有close()

```
// 每次 socket() accept() open() 都会新增一个fd
int accept(int sockfd, struct sockaddr *addr, int *addrlen);
int clientfd = accept(serverfd, NULL, NULL);
// 即便 handleClient() 内部 close() 了，主线程仍然泄漏了一个副本。
pthread_create(&tid, NULL, handleClient, (void *)clientfd);
// 修改：需要在主线程关闭副本
close(clientfd);
```

- 未检查返回值（Silent Failure）

```
/* 失败可能原因
* socket()：无可用 FD
* bind()：地址被占用
* connect()：服务器不可达
* read()/write()：对端关闭连接，网络错误
* accept()：资源不足
*/
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
// 错误
bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));
listen(sockfd, 5);
// 修改
if (sockfd < 0) {
   perror("socket");
   exit(1);}
if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
   perror("bind");
   exit(1);}
```

Linux 进程与 IPC

- fork 创建一个子进程。一个进程拥有自己的内存空间和文件描述符FD，子进程会复制父进程的整个状态（代码、全局变量、堆、栈、文件描述符），但有独立的进程标识符 PID 和内存空间

```
/*
* fork() 返回值：
* -1: 父进程在执行，fork 失败
* 0: 子进程在执行，表示当前执行的是子进程
* >0: 父进程在执行，返回值是子进程的 PID
*/
```

```
pid_t pid = fork();
if (pid < 0) {
    // fork 失败
} else if (pid == 0) {
    // 子进程
} else {
    // 父进程
}

int serverfd = socket(...);
bind(serverfd, ...);
listen(serverfd, ...);
// 服务器主循环：不断接受新客户端连接
while (1) {
    // 父进程在 accept() 处阻塞，直到有客户端连接到来。
    // 内核为该连接创建一个新的已连接套接字，并返回 clientfd。
    int clientfd = accept(serverfd, ...);
    // fork() 创建一个子进程，父子进程都会继续执行下面的代码。
    // 子进程用于处理该客户端；父进程继续接受下一个连接。
    pid_t pid = fork();
    if (pid == 0) {
        // 子进程：只负责处理这个 clientfd。
        // 此进程不再需要 serverfd（监听 socket），但至少要先处理 clientfd。
        handle(clientfd);
        // 子进程完成后终止，否则会继续执行父进程逻辑（危险）。
        exit(0);
    }
    // 父进程：不处理该客户端，因此关闭自己的那份 clientfd。
    // serverfd 继续作为监听 socket 等待下一个客户端连接。
    close(clientfd);
}
```

- exec...：进程执行到exec后，就变成全新的exec的程序，进程本身后面的代码不再执行。在子进程中应用sxec，可以让子进程单独用指定程序处理连接，而不改变父进程逻辑

```
pid_t pid = fork();
if (pid == 0) {
    // 要执行的程序路径，程序名称，程序命令行参数，NULL（结束标志）
    execl("./FileWorker", "FileWorker", filename, NULL);
    perror("exec failed");
    exit(1);
}
```

- return 和 exit() 的区别：
  - return 终止当前函数，不会结束线程。在 main 中 return 等价于 exit()
  - exit() 终止整个进程，不会继续执行任何代码
- signal() 注册处理函数（Signal Handling）：signal() 用来让程序指定一个信号发生时应执行的函数

```
// signum：要处理的信号编号，handler：触发时调用的函数
signal(int signum, void (*handler)(int));
```

- 僵尸进程：子进程已经 Terminated 结束，但父进程没有 Retrieved 回收它的退出状态，导致 Kernel 内核 Retain 保留它的PID和exit信息，occupying the PID without releasing it 占着 PID 不释放（可能导致系统无法创建新进程）。避免僵尸进程：

```
// 修复前面代码中的僵尸进程
// 1.使用 wait 或 waitpid（waitpid 是添加参数版的wait，可以指定PID，是否阻塞父进程）
while(1){
    ...
    close(clientfd);
    // close 后调用 waitpid（WNOHANG表示不阻塞父进程）
    waitpid(-1, NULL, WNOHANG);
}

// 2.父进程忽略 SIGCHLD
int main(){
    // main 开头忽略 SIGCHLD （SIG_IGN 表示 IGNore）
    signal(SIGCHLD, SIG_IGN);
    while(1){...}
}
```

- 出现僵尸进程
  - 调用了 fork()，但子进程结束后父进程没有调用 wait()/waitpid()

- wait() 写进了子进程中(无意义)

Pipe(Anonymous Pipe 匿名管道)FIFO(命名管道)

由于不同进程之间完全隔离，不能直接访问变量或内存，因此使用管道让两个进程交换数据。管道是单向的(half-duplex 半双工)

- Pipe 匿名管道:

```
// fd[0] 用于读取，fd[1] 用于写入
int fd[2];
pipe(fd)
pid_t pid = fork();
if (pid == 0) {
  // 子进程不需要写端，否则读不到EOF会一直阻塞卡住
  close(fd[1]);   // 关闭写端
  char buf[100];
  // 对方没有数据可读时，read()会一直阻塞卡住
   // read 返回值： >0 读取到的字节数；0：EOF（所有写端都被关闭）。-1：出错
  read(fd[0], buf, 100);
} else {
  // 父进程不需要读端
  close(fd[0]);   // 关闭读端
  // 对方读端关闭后继续写会触发SIGPIPE信号，导致进程退出
  write(fd[1], "hello", 5);
}
```

- FIFO 命名管道:

```
int fd;
pid_t pid;
// 参数：FIFO路径，权限
mkfifo(FIFO_PATH, 0666);
pid = fork();
if (pid == 0) {
   // 子进程读
   fd = open(FIFO_PATH, O_RDONLY);
   char buf[100];
   read(fd, buf, 100);
   close(fd);
} else {
   // 父进程写
   fd = open(FIFO_PATH, O_WRONLY);
   write(fd, "hello", 5);
   close(fd);
}
```

- 区别

Pipe	FIFO
匿名	命名
父子进程(情缘关系)，Undirectional 单向	任意两个进程，单向
进程结束即消失	名字留在文件系统重

参考题目Q1

- 在 **consumer** 函数中，为什么要在访问 ***cubbyhole** 之前调用 **pthread_mutex_lock(&mutex)**？
  加锁是为了保护临界区，避免多个线程同时访问 cubbyhole 或修改 available，防止数据竞争。

  > The mutex protects the critical section so that only one thread can access the cubbyhole or modify `available` at a time.

- **available** 变量的作用是什么？它是如何配合 **pthread_cond_wait** / **pthread_cond_signal** 一起工作的？
  available 是生产者与消费者之间的状态标记，用来表示 cubbyhole 是否有数据可取。消费者在 available=false 时等待，生产者在 available=true 时等待，二者通过 wait/signal 协调。

  > `Available` is the predicate indicating whether the cubbyhole contains data. It coordinates producer/consumer behaviour together with `pthread_cond_wait` and `pthread_cond_signal`.

- **wait** 为什么必须在 **while** 循环内调用:
  当 wait 返回后，虽然有可能是 signal 触发，但也有可能是下面两种，所以要重新检查条件。
  - Spurious Wakeup 虚假唤醒:即使没有 signal，线程也可能被唤醒
  - 条件在被唤醒时已经不成立(Race Condition 竞态条件)
  - 多个线程同时等待时，唤醒的线程不一定拿到资源

Because the thread must re-check the condition each time it wakes up. This is required due to:

1. Spurious wakeups: a thread may return from pthread_cond_wait without receiving any real signal.
2. Race conditions: after being signalled and waking up, another thread may change the shared state before the awakened thread resumes execution.
3. Multiple waiting threads: if several threads wait on the same condition, the awakened thread may not actually be the one that should proceed; thus, it must verify the condition again.

- 如果删除了生产者的 **pthread_mutex_lock** 与 **pthread_mutex_unlock**，会发生什么？

1. pthread_cond_wait 的前提条件是当前线程已经持有锁，如果生产者不加锁，则消费者的 wait 在没有锁的情况下调用是未定义行为。
2. 竞态条件：共享数据可能同时被生产者和消费者访问，而没有互斥量保护，输出顺序和读取值都可能不符合“先生产再消费的逻辑”

1. pthread_cond_wait requires the calling thread to hold the associated mutex. If the producer does not acquire the mutex in put(), then calling pthread_cond_wait without holding the lock results in undefined behaviour according to POSIX.
2. Race condition: The shared variables may be accessed concurrently by the producer and the consumer without mutual exclusion. This leads to data races, meaning the output order and the read/write behaviour may violate the logical “produce-before-consume” sequence.

- 参考代码 **producerconsumertest.c** 的 **main** 的使用方式有什么问题？可能会导致什么后果？写出的 **main**，让两个线程都被正确 **join**

1. 使用同一个 `pthread_t tid` 来保存两个不同线程的 ID，第二次 `pthread_create` 会覆盖第一次保存的 ID，且只对这个单一变量调用了一次 `pthread_join`。
2. 只能 join 到最后一个，第一个从未 join，可能造成资源泄露
3. 见参考代码

1. A single `pthread_t tid` variable is used for both threads. The second `pthread_create` overwrites the thread ID of the first thread, and there is only one `pthread_join` on this shared variable.
2. Only the last created thread is joined, the first one is never joined, which may cause a resource leak.

- 分析下列三种情况下是否会出现死锁：

在正确的流程中，生产者在 **`put()`** 里：加锁 → **while** 等待 → 写入 **value**、设置 **hasValue = true** → **signal** → 解锁，消费者在 **`get()`** 里：加锁 → **while** 等待 → 读取 **value**、设置 **hasValue = false** → **signal** → 解锁，生产者生产次数是固定的

- 程序只创建并启用一个生产者线程，不创建任何消费者线程，生产者会循环调用 **put()** 生产若干值后退出
- 程序创建两个生产者线程和一个消费者线程共享同一个**CubbyHole**缓冲区
- 程序创建一个生产者线程和两个消费者线程共享一个缓冲区

1. 初始状态下 hasValue == false，不会进入 while，但是操作并设置为 true 之后，再次执行会因为 true 而进入 while，调用 wait。而因为没有人唤醒它，所以会永远等待
2. 当缓冲区空（hasValue == false）时，某个生成者会成功写入一个值并设置 true，而另一个生产者再看会因为 true 而 wait。消费者消费一次后，变成 false，唤醒 wait 的生产者，因此线程安全不会死锁
3. 生产者只按固定次数生产，生产完就退出；两个消费者总消费次数大于生产次数，因此在缓冲区变为空之后没有线程再 signal，导致消费者永远等待。

1. Initially, hasValue == false, so the producer does not enter the while loop. After producing and setting it to true, the subsequent execution will see true and enter the while, calling wait. Since no one wakes it up, it will wait forever.
2. When the buffer is empty (hasValue == false), one producer will successfully write a value and set it to true. The other producer will see the truth and go into wait. After the consumer consumes the value, it becomes false again, and the consumer wakes the waiting producer. Therefore, the system is thread-safe and will not deadlock.
3. The producer generates only a fixed number of items and then exits. The two consumers together attempt to consume more items than were produced. Once the buffer is empty, no thread will perform a signal again, causing consumers to wait forever.

- 假设 **`CubbyHole`** 被多个生产者和多个消费者共享，大家都在同一个条件变量 **`cubbyCond`** 上等待和唤醒。

- 在什么情况下你会考虑把 **`pthread_cond_signal`** 改成 **`pthread_cond_broadcast`**？（什么场景下只唤醒一个线程不够，必须唤醒所有等待线程？）
- 如果总是 **broadcast** 而不是 **signal**，可能会带来哪些问题？（性能+逻辑）

1. 当有多种类型的线程都在等同一个条件，条件变化对所有等待线程都可能有影响时，可能需要多个线程同时重新判断各自的条件。例如全局状态改变，或者通知所有在 wait 的线程退出等。
2. 性能问题：频繁 broadcast 会唤醒大量不需要继续执行的线程，造成多次抢锁和上下文切换，性能下降。
   逻辑问题：broadcast 可能让很多线程同时醒来、一起抢资源，导致逻辑上的抖动或某些线程长期饥饿。

1. When multiple threads of different types are waiting on the same condition, and a change to that condition may affect all waiting threads, it may be necessary to wake all of them so they can re-evaluate their own conditions. Examples include global state changes or notifying all waiting threads to exit.

2. Performance issue: Frequent broadcasts may wake many threads that do not need to run, causing repeated lock contention and context switches, degrading performance.
   Logical issue: A broadcast may wake many threads at the same time, causing them to compete for resources simultaneously. This can lead to logical instability (thrashing) or starvation of specific threads.

- 如果把 **`available`** 的初始值改成 **`true`** 会怎么样？
- 如果有多个生产者线程和多个消费者线程同时使用同一个 **`cubbyhole`**，当前使用 **`pthread_cond_signal`** 的实现可能出现什么问题？请给出一个**“可能导致线程永远等待或逻辑错误”**的高层执行场景描述。

1. 第一次消费者会读到未初始化值
2. 由于只有一个条件变量和一个布尔状态，生产者/消费者 signal 时可能唤醒另一个不满足状态的同类消费者/生产者，导致程序一直等待

1. The consumer will read an uninitialized value the first time.
2. Because there is only one condition variable and a single boolean state, a signal from a producer or consumer may wake up another thread of the same type that does not meet the required condition, causing the program to wait indefinitely.

参考题目**Q2**

- **`socket()`** 和 **`connect()`** 的作用

  `socket()`：在内核中创建一个 TCP 套接字，返回文件描述符，用于后续读写和连接。

  `connect()`：把这个套接字主动连接到指定服务器 IP 和端口，完成 TCP 三次握手，建立客户端到服务器的连接。

The `socket()` call creates a TCP socket in the kernel and returns a file descriptor used for I/O.
The `connect()` call actively connects this socket to the given server IP and port, completing the TCP handshake and establishing the client–server connection.

- 在 **`connect`** 失败后直接调用 **`exit(1)`** 存在什么资源管理问题？如何修复？

  `connect` 失败时直接 `exit(1)` 没有先 `close(sock)`，会导致套接字文件描述符泄露。应先 close(sock) 关闭套接字

On `connect` failure, the code calls `exit(1)` without first closing `sock`, which leaks the socket descriptor. It should close the socket before exiting.

- 程序最后在子进程中调用 **`exit(0)`**，而不是 **`return 0;`**，两者有什么区别？在子进程里使用 **`exit`** 是否合理？

  在 `main` 中 `return 0;` 会从 `main` 返回并析构其局部对象，而 `exit(0)` 直接终止进程、运行 `atexit` 处理和静态析构但不返回到 `main`。在子进程里用 `exit(0)` 表达“立即以状态 0 结束整个子进程”是合理的。

In `main`, `return 0;` returns from `main` and destroys its local objects, while `exit(0)` terminates the process directly after running `atexit` handlers and static destructors.
In this child process case, using `exit(0)` to end the whole child with status 0 is reasonable.

- 如果要保证子进程退出前线程正确结束，请给出一个合理的关闭顺序。

  先设置停止标志（如 `stopRequested = true`），让下载循环和进度线程都能感知；然后对子线程调用 `pthread_join` 等待其退出；最后依次关闭文件 `fclose(fp)` 和套接字 `close(sock)`，再 `exit` 或 `return`。

First, set a stop flag (e.g. `stopRequested = true`) so both the download loop and the progress thread can break out.
Then call `pthread_join` on the progress thread, and only afterwards close the file and socket, and finally exit the child.

- 为什么必须循环调用 **`read`** 并根据返回值 **`n`** 精确处理，而不能假设**“一次 `read` 就拿到全部想要的数据”**？

  因为 TCP 是字节流，不保证应用层一条消息对应一次 `read`：可能被拆成多次 `read`，也可能几条消息合并在一次 `read` 里。只有循环读取并按每次返回的 `n` 精确处理、累积，才能保证不丢数据、不粘包。

TCP is a byte stream and does not preserve message boundaries. A single logical message may be split across multiple `read`s or combined with other messages in a single `read`.
You must therefore loop on `read` and handle exactly `n` bytes each time to avoid data loss or message framing errors.

- **`stopRequested`** 是普通 **`bool`**，在信号处理函数和其他线程间共享，可能存在哪些可见性和原子性问题？

  普通 `bool` 既不是 `sig_atomic_t` 也没有任何同步，信号处理函数和其他线程并发读写它会产生数据竞争：赋值可能不是原子的，其他线程可能看不到最新值（缓存、重排序），行为依赖编译器和平台。

A plain `bool` is neither `sig_atomic_t` nor protected by any synchronization.
Concurrent access from the handler and other threads creates data races: updates may not be atomic and other threads may not see the latest value, leading to implementation-dependent behaviour.

- 如果将来可能 **`fork`** 多个子进程，你会如何调整 **`wait`** 的使用以避免僵尸进程**?**

  应当循环调用 `wait` 或 `waitpid`：用 `while ((pid = wait(&status)) > 0) { ... }` 回收所有子进程，或使用 `waitpid(childPid, &status, 0)` 针对特定子进程。否则只 `wait` 一次会留下其他子进程成为僵尸。

You should loop on `wait` or `waitpid`, e.g. `while ((pid = wait(&status)) > 0) { ... }`, or call `waitpid` for

specific PIDs, so that all children are reaped, and no zombies remain.

- 对比两种设计：在子进程中直接写下载逻辑；**fork** 后在子进程中 **exec** 独立的 **filedownloader** 程序。各自优缺点是什么？
  1. 直接在子进程写逻辑：实现简单，能直接访问父进程的代码和数据；缺点是父子高度耦合，共享太多状态，父进程改动容易破坏子进程假设。
  2. `fork + exec`：子程序有全新的地址空间和清晰的接口（通过参数或 IPC 通信），模块边界更干净；缺点是参数和 IPC 设计更复杂，部署和调试成本略高。

  1. Direct logic in the child: easy to implement and can freely access the parent's code and data, but tightly couples parent and child and shares a lot of state.
  2. `fork` + `exec`: runs a fresh program with a clean address space and a clear interface via arguments or IPC, improving modularity but adding complexity in argument passing and inter-process communication.

- 使用 **`execvp`** 或 **`execlp`** 时，如果调用失败会发生什么？代码中应该如何处理这种失败？

  `execvp`/`execlp` 如果成功不会返回；若失败返回 -1 并设置 `errno`。因此在调用后应立即检测失败、输出错误并用 `_exit` 结束子进程，例如：

```
execlp("filedownloader", "filedownloader", /* args */, (char*)NULL);
perror("execlp failed");
_exit(1);
```

If `execvp`/`execlp` succeeds, it never returns; on failure, it returns -1 and sets `errno`.
So immediately after the call you should handle failure, e.g.:

参考代码**Q1**

producerconsumertest.c

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

static pthread_mutex_t mutex;
static pthread_cond_t cond;
static bool available = false;

void * consumer(void * arg) {
 int* cubbyhole = (int *) arg;
 for (int i= 0; i < 10; i++) {
    pthread_mutex_lock(&mutex);
    while(!available)
       pthread_cond_wait(&cond, &mutex);
    printf("get %d\n", *cubbyhole);
    available = false;
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond);
 }
}
void * producer(void * arg) {
 int*  cubbyhole = (int *) arg;
 for (int i= 0; i < 10; i++) {
    pthread_mutex_lock(&mutex);
    while(available)
       pthread_cond_wait(&cond, &mutex);
    *cubbyhole = i;
    printf("put %d\n", *cubbyhole);
    available = true;
    pthread_mutex_unlock(&mutex);
    pthread_cond_signal(&cond);
 }
}

main() {
 pthread_t tid;
```

```
 int *ret;
 int cubbyhole;
 pthread_create(&tid, NULL, producer, (void*) &cubbyhole);
 pthread_create(&tid, NULL, consumer, (void*) &cubbyhole);
 pthread_join(tid, (void **)&ret);
}

// 修改后的 main
int main() {
   pthread_t prod_tid, cons_tid;
   int cubbyhole;
   pthread_create(&prod_tid, NULL, producer, &cubbyhole);
   pthread_create(&cons_tid, NULL, consumer, &cubbyhole);
   pthread_join(prod_tid, NULL);
   pthread_join(cons_tid, NULL);
   return 0;
}
```

Thread.h

```
#pragma once
class Thread {
public:
   Thread(Thread *childThread);
   ~Thread();
   void run() {}
   void start();
private:
   void *state;
   Thread *childThread;
};
```

Thread.cpp

```
#include "Thread.h"
#include "SimpleThread.h"
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
void* startMethodInThread(void *arg)
{
   if (arg == NULL)
      return 0;
   SimpleThread *thread = (SimpleThread*)arg;
   thread->run();
   return NULL;
}
Thread::Thread(Thread *childThread) {
   this->state = malloc(sizeof(pthread_t));
   this->childThread = childThread;
}
void Thread::start() {
       pthread_t tid;
   pthread_create(&tid, NULL, startMethodInThread, (void *) this);
   memcpy(this->state, (const void *)&tid, sizeof(pthread_t));
}
Thread::~Thread() {
   free(this->state);
}
```

参考代码Q2

fork, exec, waitpid

```
pid_t pid = fork();
```

```
if(pid == 0){
   execl("/bin/ls","ls","-l",(char*)NULL);
   _exit(1); // exec失败才会走到这里
}

int status;
waitpid(pid,&status,0);
```

pthread + socket

```
void* threadFunc(void*){
   char buf[1024];
   int n = read(sock, buf, sizeof(buf));
   return NULL;
}
```

socketserver

```
int main() {
 int sock, length;
 struct sockaddr_in server;
 int msgsock;
 char buf[1024];
 int rval;
 int i;

 // ====== 服务器初始化流程 Step 1: 创建 socket ======
 // AF_INET       表示使用 IPv4
 // SOCK_STREAM   表示使用 TCP（面向连接的字节流）
 // 第三个参数 0 让系统自动选择协议（对 AF_INET + SOCK_STREAM 一般就是 TCP）
 sock = socket (AF_INET, SOCK_STREAM, 0);
 if (sock < 0) {
   perror("opening stream socket");
 }

 // ====== 服务器初始化流程 Step 2: 填写本地地址信息（server sockaddr_in） ======
 // 设置地址族为 IPv4
 server.sin_family = AF_INET;
 // INADDR_ANY 表示绑定到本机所有网卡的地址（0.0.0.0）
 server.sin_addr.s_addr = INADDR_ANY;
 // 这里直接写 8888，严格来说应该用 htons(8888) 转成网络字节序
 server.sin_port = 8888;

 // ====== 服务器初始化流程 Step 3: bind 把 socket 和 IP:port 绑定起来 ======
 // 把 sock 这个套接字绑定到 server 指定的本地地址和端口
 if (bind (sock, (struct sockaddr *)&server, sizeof server) < 0) {
   perror ("binding stream socket");
 }

 // ====== 服务器初始化流程 Step 4: listen 把 socket 变成“监听套接字” ======
 // 第二个参数 5 表示“等待连接队列”的最大长度
 listen (sock, 5);

 // ====== 服务器初始化流程 Step 5: accept 等待客户端连接（阻塞） ======
 // accept 会阻塞，直到有客户端来连
 // 返回的新 socket（msgsock）专门用于和该客户端通信
 msgsock = accept(sock, (struct sockaddr *)0, (socklen_t *)0);
 if (msgsock == -1) {
   perror("accept");
 }

 // ====== 建立连接后的通信：从客户端读数据 ======
 // 从 msgsock 中读取最多 1024 字节数据到 buf
 // read 也可能阻塞，直到客户端发送了数据或者关闭连接
 if ((rval = read(msgsock, buf, 1024)) < 0){
   perror("reading socket");
 } else  {
   printf("%s\n",buf);
 }

 // ====== 通信结束后关闭和客户端对应的 socket ======
```

```
 close (msgsock);
}
```

socketclient

```
int main() {
 int sock;
 struct sockaddr_in server;
 int msgsock;
 char buf[1024];
 struct hostent *hp;
 char *host = "127.0.0.1";
 int rval;

 // ====== 客户端初始化流程 Step 1: 创建 socket ======
 // 和服务器一样，创建一个 IPv4 + TCP 的套接字
 sock = socket (AF_INET, SOCK_STREAM, 0);
 if (sock < 0) {
   perror("opening stream socket");
 }

 // ====== 客户端初始化流程 Step 2: 准备服务器地址结构体 ======
 // 先把 server 结构清零，避免有脏数据
 bzero(&server, sizeof(server));

 // 通过主机名（这里是 "localhost"）查询 IP 地址
 // 结果存在 hostent 结构里
 hp = gethostbyname("localhost");

 // 把查询到的 IP 地址拷贝到 server.sin_addr 中
 bcopy((char*)hp->h_addr, (char*)&server.sin_addr, hp->h_length);

 // 设置地址族为 IPv4
 server.sin_family = AF_INET;

 // 设置要连接的服务器端口号
 // 服务器那边 bind 了 8888，这里也要连 8888
 // 严格写法也应该使用 htons(8888)
 server.sin_port = 8888;

 // ====== 客户端初始化流程 Step 3: connect 主动发起连接 ======
 // 客户端用 connect() 向 server 指定的 IP:port 发起 TCP 连接
 // 成功后，sock 就代表一条已经建立好的 TCP 连接
 if (connect(sock, (struct sockaddr*)&server, sizeof(server))<0){
   perror("connecting");
 }

 // ====== 建立连接后的通信：先向服务器发送数据 ======
 // 这里把字符串 "/usr/include" 拷贝到 buf 中
 strcpy(buf,"/usr/include");

 // 把 buf 的内容通过 socket 写到服务器
 // strlen(buf)+1 包含结尾 '\0'
 if ((rval = write(sock, buf, strlen(buf)+1)) < 0){
   perror("writing socket");
 }

 // ====== 从服务器读回数据并打印 ======
 // 不断从 sock 里读取数据到 buf
 // read 返回 0 表示对方关闭连接，循环结束
 while (read(sock, buf, sizeof(buf)) > 0){
   printf("%s", buf);
 }

 // ====== 通信结束后关闭 socket ======
 close (sock);
}
```