中国科学技术大学

多线程程序设计

徐 伟

E-mail: xuweihf@ustc.edu.cn

课程概述

- 线程与进程
- 线程的分类
- 最简单的多线程程序
- 线程同步方式
- 线程相关函数
- 相关实例

进程实例： Hello World 从源码到进程

第一步：写源码

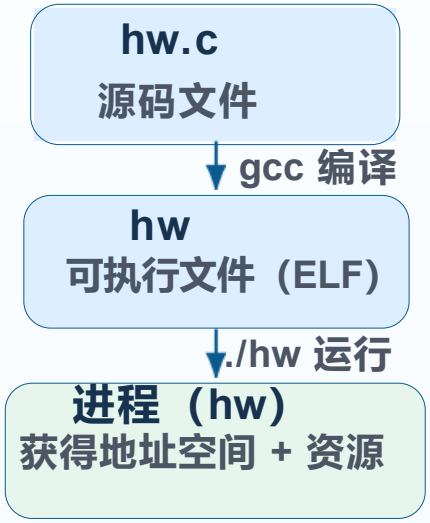
```
/* hw.c */
#include <stdio.h>
int main(void) {
  while(1){;}
  return 0;
}
```

第二步：编译成可执行文件

```
gcc hw.c -o hw
```

第三步：运行操作系统创建进程

```
./hw
```



关键区别

hw（硬盘上的文件）是程序；
运行后在内存里的实例才是进程。

hw.c

用 ps 命令查看进程

常用 ps 命令

```
# 查看当前终端的进程
ps

# 查看所有进程（最常用）
ps -a

ps -aux | grep hw
```

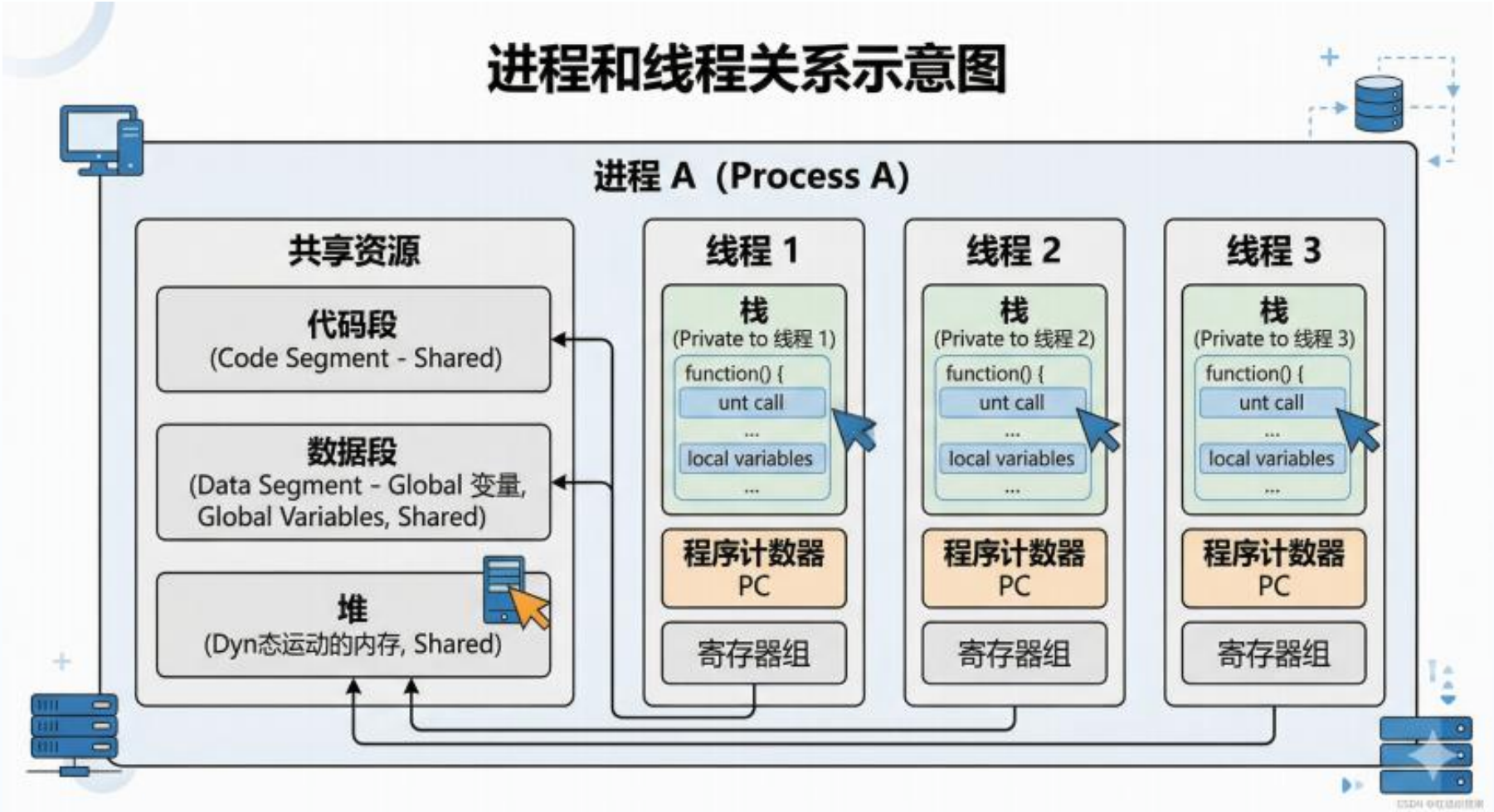
字段	含义
USER	启动该进程的用户
PID	进程 ID， 操作系统给每个进程分配的唯一编号
%CPU	CPU 占用百分比
%MEM	内存占用百分比
COMMAND	进程对应的命令名

示例输出（ps aux节选）

在一个终端运行 ./hw

另一个终端执行 ps -aux | grep hw 查看进程

```
xxxwww      2203  0.3  0.7 420372 31560 ?        Sl   14:17   0:00 update-no
root        2226  0.0  0.0   2616    96 ?        S    14:17   0:00 /bin/sh /
root        2228  0.0  0.0   8084   576 ?        S    14:17   0:00 sleep 174
xxxwww      2398  103  0.0   2364   576 pts/0    R+   14:18   0:10 ./a.out
xxxwww      2401  0.3  0.1  10616  4864 pts/1    Ss   14:18   0:00 bash
xxxwww      2408  0.0  0.0  11688  3524 pts/1    R+   14:18   0:00 ps -aux
xxxwww@ubuntu:~/Desktop/Pro$
```



一个进程至少包含一个线程（主线程）

线程共享：代码段、全局/静态区、堆、打开的文件

线程私有：栈、程序计数器、寄存器现场

线程与进程

- 进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配的基本单位，是操作系统结构的基础。在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体

- 线程（thread）是操作系统能够进行运算调度的最小单位。它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务

线程与进程

进程 Process

- 一个正在运行的程序实例
- 操作系统给它分配地址空间和资源
- 不同进程默认相互隔离

线程 Thread

- 进程内部的一条执行流
- 同一进程内可以有多条线程
- 多条线程共享进程的大部分资源

一句话

进程和线程关系是一对一，或一对多

进程像“运行中的程序容器”，线程像“容器里真正执行任务的工人”。

任务（task）和线程之间的关系
可以是一对多或多对一

线程的分类

- 主线程
  - 当一个进程被启动时，操作系统自动创建的第一个线程，即程序入口点（如C的main()函数）所在的线程
  - 作用：
    - 执行程序初始化、创建其他工作线程

线程的分类

- 子线程（工作线程/用户线程）
  - 由主线程或其他工作线程在程序运行过程中显式创建出来的线程
  - 作用：
    - 分担主线程的任务，如执行耗时计算、I/O操作、后台服务等，以提高并发度和响应性
  - 特点：
    - 拥有独立的执行流和栈空间。
    - 可以设置是否为守护线程（即是否随主线程退出而自动结束）。
    - 工作线程结束后，其资源需要被回收

线程的分类

- 其他分类方式
- 按功能
  - I/O线程、计算线程、定时器线程、网络线程等
- 按调度实现
  - 用户级线程、内核级线程、混合线程
- 最简单的二分法：**主线程 + 工作线程**

本课只用最简单的二分法

主线程 + 工作线程

为什么这样分?

- 直接对应代码中”谁创建谁”的关系，方便理解生命周期。
- 其他分法属于扩展进阶知识，用到特定场景时再去查阅即可。

线程分类：入门阶段先抓住主线程和工作线程

主线程

- 进程启动后系统创建的第一个线程
- 程序入口通常是 main()
- 负责初始化、创建其他线程、最后收尾

工作线程（子线程）

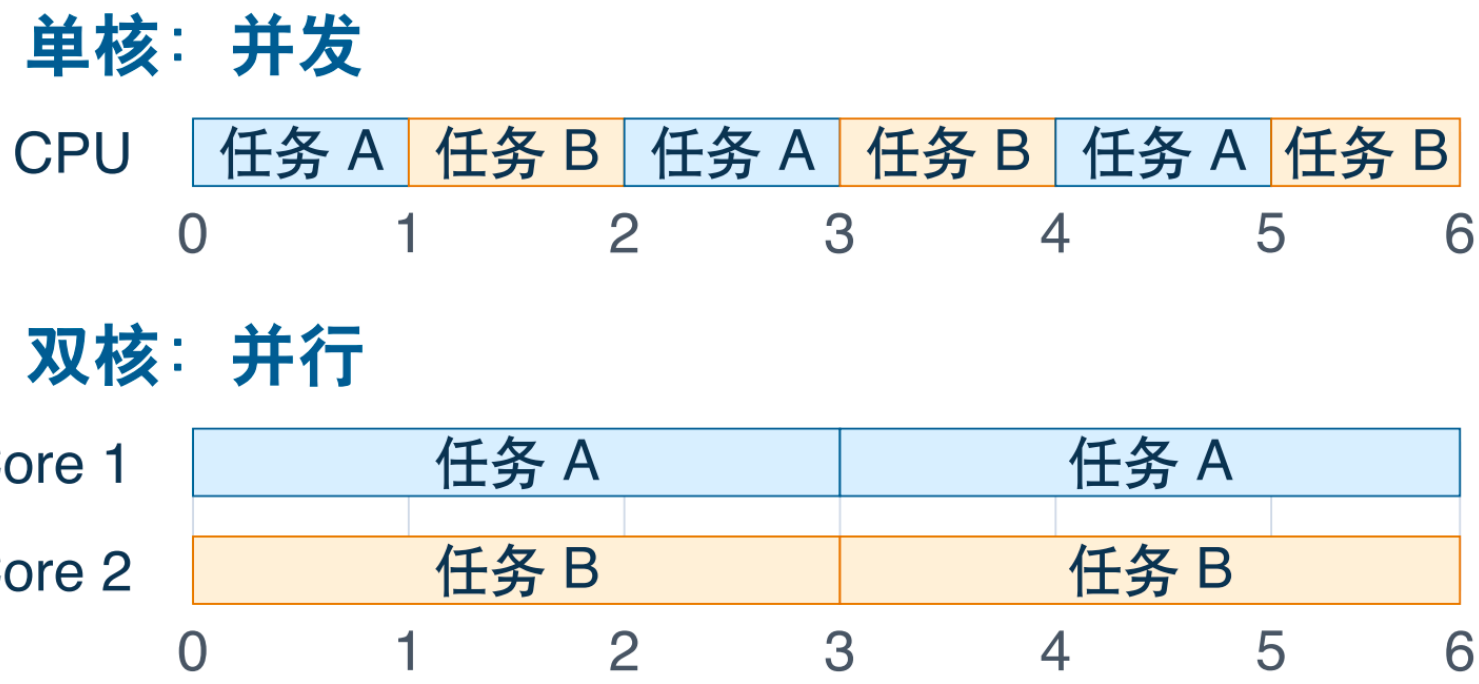
- 由主线程在运行时手动创建
- 帮主线程干耗时活，防止程序卡死
- 有独立的执行流，互不干扰
- 结束时需资源回收：要么让主线程 join，要么设为detached

线程与进程

- 程序的并发与并行
- 并发Concurrency
  - 多个任务在同一时间段内推进
  - 可能只是交替执行
  - 单核上也能出现并发
- 并行 Parallelism
  - 多个任务在同一时刻真正执行
  - 通常依赖多核或多处理器
  - 更容易带来 CPU 计算提速
- 并发强调“**同时推进**”，并行强调“**同时执行**”

并发和并行示意图

- 单核机器上的多线程，优势常来自隐藏等待时间；多核机器上的多线程，才更容易得到真正的计算提速



多线程的收益

- 提高响应性：在 GUI 或网络服务中，主线程处理用户交互，工作者线程执行耗时操作（如文件 I/O、网络请求），避免界面“卡死”
- 充分利用多核 CPU：将计算密集型任务分解到多个线程，实现真正的并行计算，缩短总执行时间
- 资源高效利用：线程共享进程资源，创建和切换开销远低于进程，适合高并发场景

思考题

- 增加多线程的优势/收益
- 在单核上，用多线程有优势吗?
- 在多核上，用多线程有优势吗?

增加多线程的优势/收益。

- 问题 1. 在单核机器上，用多线程有优势吗?
  - 只有一个 CPU 核心，所有线程其实还是在交替运行，那用多线程图什么?
- 问题 2. 在多核机器上，用多线程有优势吗?
  - 有多个核心了，是不是任何代码挂上多线程都会变快?

问题 1. 在单核机器上，用多线程有优势吗？

- 答：对于I/O密集型任务，多线程有明显优势，主要为了“防卡死”
  - 利用等待：某线程等I/O时，CPU切给另一线程
  - 提高响应：工作线程后台干活，主线程即时响应→没算得更快，但挤出了闲置等待时间

- 对于计算密集型任务，多线程没有优势甚至更差

问题2:在多核机器上，用多线程有优势吗?

- 答：对于I/O密集型任务，能实现“真加速”

  对于计算密集型任务，多线程有显著优势

  - 真正并行：大任务拆分给多核“同时”算
  - 最适合极少共享资源的CPU密集任务

最简单的多线程程序

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>  // for sleep()


void* thread_func(void* arg) {
    // 子线程执行的函数
    int id = *(int*)arg;
    printf("Worker %d: Hello!\n", id);
    pthread_exit(NULL); // 子线程主动退出
}

int main() {
    pthread_t t1;
    int arg1 = 1;

    // 创建一个子线程
    pthread_create(&t1, NULL, thread_func, &arg1);

    printf("Main thread: waiting...\n");
    sleep(1);
    pthread_exit(NULL); // 主线程退出，系统等剩余线程结束后才终止进程
}
```

02easy_pthread.c

最简单的多线程程序

编译命令

gcc 02easy_pthread.c -lpthread

- 若主线程最后写return 0;
- 会立即终止整个进程（子线程也死掉）
- 改用pthread_exit(NULL)可以让主线程安全退出，而进程继续存活直到全员完工

03easy_pthread.c

最简单的多线程程序

- pthread_create四个参数是什么?
  - //pthread_create(&t1,NULL,thread_func,&arg1);
  - //[1][2][3][4]

#	参数	含义
[1]	&t1	t1把新线程ID。先声明pthread_t t1 &t1传地址
[2]	NULL	线程属性（栈大小、优先级……）。入门写NULL即可
[3]	thread_func	子线程执行的函数。函数名必须是void*f(void*)
[4]	&arg1	传给thread_func的参数。不传参数写NULL

最简单的多线程程序

- pthread_exit的参数与作用
  - void pthread_exit(void*retval);
  - //[1]
    - retval：线程的”返回值”（void*指针），可以被pthread_join取走。不需要就写NULL
- 子线程调用——只退出自己
  - 调用后只有当前线程结束，其他线程照常运行
- 主线程调用——进程不立即结束
  - 主线程自己退出，但系统会等剩余子线程结束后才销毁进程

线程的同步方式

- 如果线程之间不同步，比如对同一个参数操作，由于没有同步产生计算/逻辑错误会发生什么？

- 错误示例：两个线程同时累加同一个计数器（上）
- 代码目的：两个线程各做100000次加一，按直觉应当输出200000；这个例子专门用来观察竞态条件

```
#include <pthread.h>
#include <stdio.h>
int counter = 0;
void* add_one(void* arg) {
    int loops = *(int*)arg;
    for (int i = 0; i < loops; ++i) {
        int tmp = counter;
        tmp = tmp + 1;
        counter = tmp;
    }
    return NULL;
}
```

04count.c

```
int main(void) {
    pthread_t t1, t2;
    int loops = 100000;

    pthread_create(&t1, NULL, add_one, &loops);
    pthread_create(&t2, NULL, add_one, &loops);

    pthread_join(t1, NULL);//调用 pthread_join(thread, ...) 的线程会阻塞，直到指定
                                        的 thread 线程执行完毕
    pthread_join(t2, NULL);

    printf("counter=%d\n", counter);

    return 0;
}
```

线程的同步方式

- 上面代码运行结果：

```
xxxwww@ubuntu:~/Desktop/Pro$ gcc 04count.c -lpthread
xxxwww@ubuntu:~/Desktop/Pro$ ./a.out
counter=166594
xxxwww@ubuntu:~/Desktop/Pro$ ./a.out
counter=113485
xxxwww@ubuntu:~/Desktop/Pro$ ./a.out
counter=150833
xxxwww@ubuntu:~/Desktop/Pro$ ./a.out
counter=136628
xxxwww@ubuntu:~/Desktop/Pro$
```

- 期望结果：输出counter = 200000
- 实际结果：不相符，而且每次运行可能不同
- 通过学线程同步，可以很好理解这个问题

线程的同步方式

- 这个程序结果为什么不相符
- 期望结果
  - 两个线程各加100000次，counter最终应为200000
- 实际结果
  - 程序输出比200000小的数，而且每次运行结果可能不同
- C 语言里 counter = counter + 1 语句，到了 CPU 层面通常被拆成三条独立的机器指令：
  - 读：从内存中把 counter 的值读到寄存器（例如 load counter, R1）
  - 改：在寄存器里加 1（add R1, 1, R2）
  - 写：把新的值写回内存（store R2, counter）

线程的同步方式

- 当两个线程同时在多核（或单核时间片轮转）上执行时，指令的执行顺序可能交错，产生类似下面的情况：

时间	线程 A 执行指令	线程 B 执行指令	counter 内存值
T1	读 counter (得 100) → 准备写		100
T2		读 counter (依然读得 100)	100
T3	计算 100+1 = 101	计算 100+1 = 101	100
T4	写回 counter = 101		101
T5		写回 counter = 101 (覆盖)	101

- 结果：两次操作，counter只增加了 1，丢失了一次更新

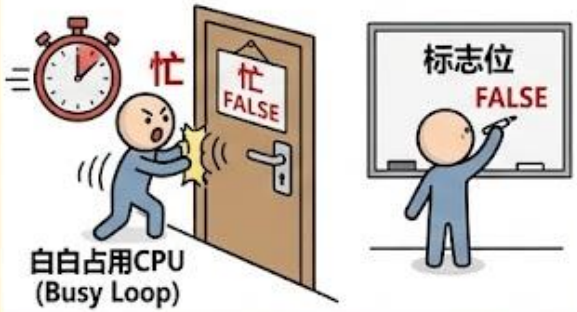
线程的同步方式

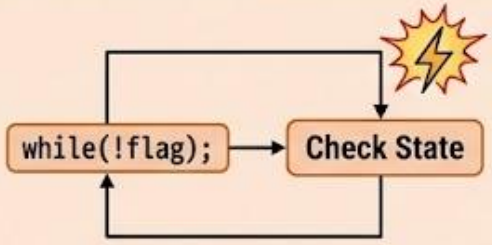
线程同步机制示意图：从朴素到高效

1. 朴素 (Naïve) / 忙等 (Busy-waiting)

共享标志位全局变量作信号。
缺点：白白占用CPU（忙等），易错。

不停查看门锁



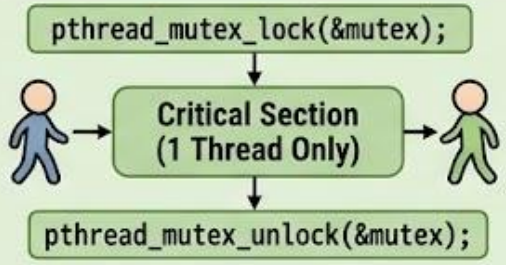


解决“等没等到”，但效率极低。

2. 互斥锁 (pthread_mutex_t)

同一时刻只准一个线程进入关键区。
解决“能不能改”的问题。

厕所占位

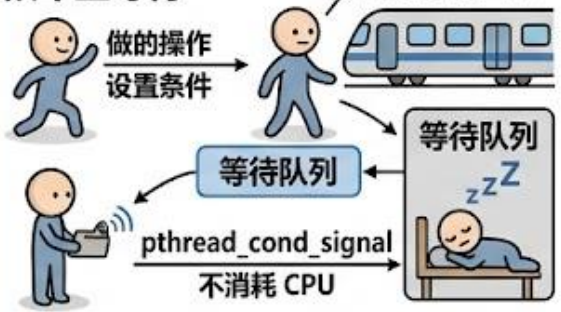


解决“能不能改”共享资源。

3. 条件变量 (pthread_cond_t)

条件不满足时睡眠，满足时再唤醒。
解决“该不该等”的问题。

候车室等待




```
pthread_mutex_lock(&mutex);
Check Condition (FALSE)
pthread_cond_wait(&cond,&mutex);
(Automatically unlock, Sleep)
(Woken Up)
Re-lock → Critical Section
pthread_mutex_unlock(&mutex);
...
... data = TRUE; → pthread_cond_signal(&cond);
```

解决“该不该等”，不消耗 CPU。

线程的同步方式-1

- 最原始的做法：共享标志位通信（不推荐使用）
  - 设置一个全局变量，通过变量值的改变进行通信。
  - 场景举例：主线程循环3次sleep后修改变量为1，子线程用while不断判断。
- 这种做法的痛点
  - 它一方面会让子线程一直空转，白白消耗CPU（忙等待/Busy Wait）；
  - 另一方面又没有用互斥锁或条件变量建立可靠同步，严格说这个例子还存在“另一个线程是否一定能及时看到ready = 1”的问题。

线程的同步方式-1

- 原始做法代码演示：共享标志位通信

工作线程（接收信号）

```c
#include <unistd.h>
int ready = 0; // 全局标志

void* worker(void* arg) {
  /* 一直死循环检查，白白消耗CPU */
  while (ready == 0) {
  }
  printf("worker: 收到信号！\n");
  return NULL;
}
```

主线程（发送信号）

```c
int main(void) {
  pthread_t tid;
  pthread_create(&tid, NULL,
      worker, NULL);
  for (int i = 0; i < 3; i++) {
    printf("main: sleep %d\n", i
        +1);
    sleep(1);
  }
  ready = 1;
  pthread_join(tid, NULL);
  return 0;
```

05busy_wait.c

```
(base) chulijie@chulijiedeMacBook-Pro code % gcc busy_wait.c -o busy_wait
(base) chulijie@chulijiedeMacBook-Pro code % ./busy_wait
main: 准备中... sleep 1 秒
worker: 等待主线程发来信号...
main: 准备中... sleep 2 秒
main: 准备中... sleep 3 秒
main: 准备完毕，将 ready 设为 1
worker: 收到信号！工作线程开始执行后续任务。
```

线程的同步方式-2

- 互斥锁的核心生命周期（类似网络套接字）
  - 互斥锁四大核心操作

1. 初始化（诞生）

分配内部结构，设置初始状态为“没被锁住”。
`pthread_mutex_init`

2. 加锁（占据）

申请通行证。如果别人抢先了，我就在这里等，直到别人放开。
`pthread_mutex_lock`

3. 解锁（释放）

用完还回去，系统会喊醒某个正在等通行证的线程。
`pthread_mutex_unlock`

4. 销毁（收尾）

不用了就彻底销毁，释放操作系统对应的资源。
`pthread_mutex_destroy`

线程的同步方式-2

- 互斥锁API详解（1）：初始化与销毁

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int  pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- mutex：要操作的那把锁的地址（需提前声明pthread_mutex_t）。
- attr：锁的属性。这和前面pthread_create一样，入门直接填NULL使用默认属性即可

对于全局锁，C语言提供了一个不需要调init和destroy的宏写法：pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
后面的代码例子都会用这种最简写法！

线程的同步方式-2

- 互斥锁API详解（2）：加锁与解锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- pthread_mutex_lock：
  - 状态：如果锁是空闲的，立刻拿锁走人，继续往下执行
  - 阻塞：如果锁已被别人拿着，当前线程会自动睡眠（不卡CPU），直到那个人解锁
- pthread_mutex_unlock：
  - 动作：把这把锁改回“空闲”状态
  - 唤醒：如果有其它线程正被卡在lock上睡觉，系统会挑一个唤醒它

线程的同步方式-2

- 千万要记住的一件事一定要成对出现！并且必须是“谁加的锁，谁负责解”

线程的同步方式-2

- 正确写法：用互斥锁保护关键区（上）
- 代码目的：仍然让两个线程各加100000次，但要求结果稳定正确。

```c
#include <pthread.h>
#include <stdio.h>
int counter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* add_one_safe(void* arg) {
  int loops = *(int*)arg;
  for (int i = 0; i < loops; ++i) {
    pthread_mutex_lock(&mutex);
    counter++;
    pthread_mutex_unlock(&mutex);
  }
  return NULL;
}
```

06count_mutex.c

```c
int main(void) {
  pthread_t t1, t2;
  int loops = 100000;
  pthread_create(&t1, NULL, add_one_safe, &loops);
  pthread_create(&t2, NULL, add_one_safe, &loops);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("counter = %d\n", counter);
  return 0;
}
```

线程的同步方式-2

- 期望结果与实际结果：都应稳定为counter = 200000，因为同一时刻只有一个线程能进入关键区。

```
xxxwww@ubuntu:~/Desktop/Pro$ gcc 06count_mutex.c -lpthread
xxxwww@ubuntu:~/Desktop/Pro$ ./a.out
counter = 200000
xxxwww@ubuntu:~/Desktop/Pro$
```

线程的同步方式-3

- 死锁示例：两把锁顺序不一致

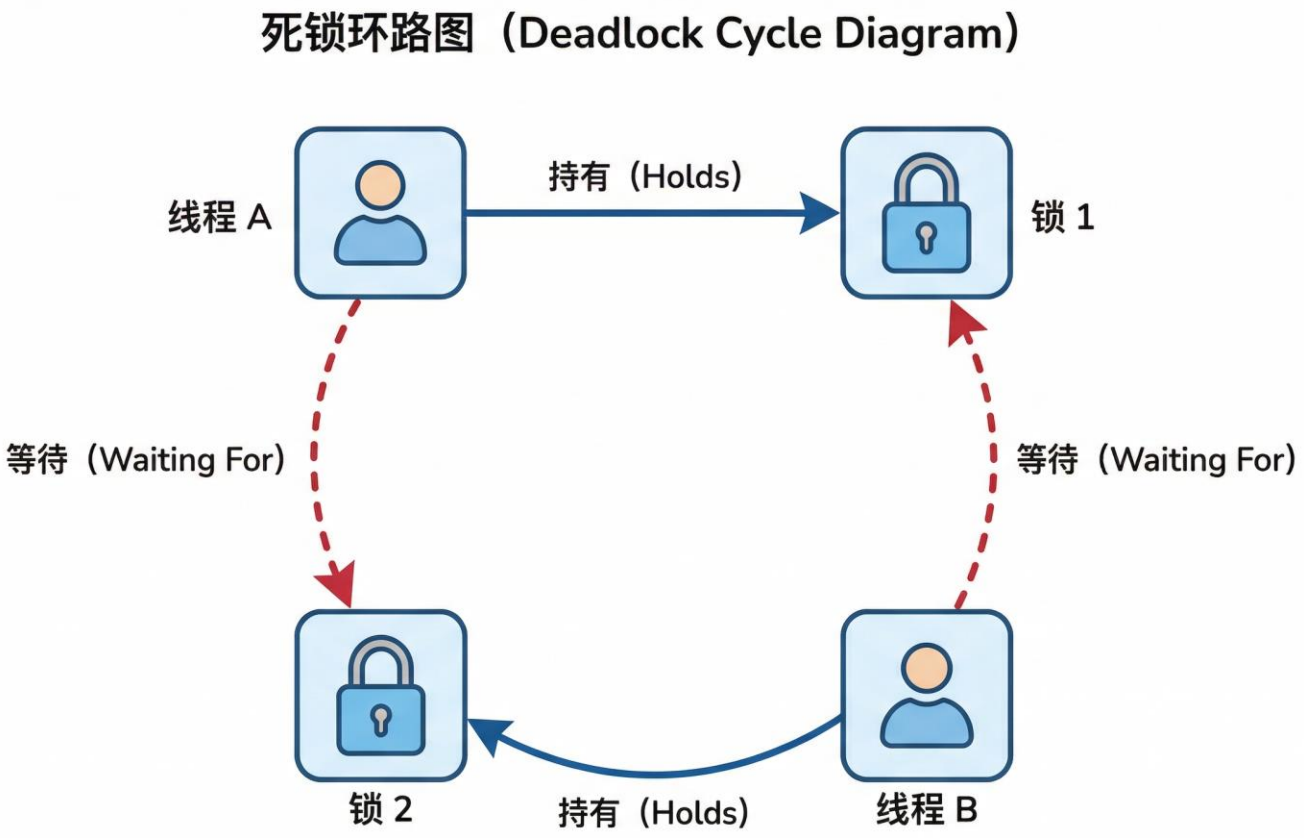
```
#include <pthread.h>
#include <unistd.h>
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
void* worker_a(void* arg) {
  pthread_mutex_lock(&m1);
  sleep(1);
  pthread_mutex_lock(&m2);
  return NULL;
}

void* worker_b(void* arg) {
  pthread_mutex_lock(&m2);
  sleep(1);
  pthread_mutex_lock(&m1);
  return NULL;
}
```

07deadlock.c

线程的同步方式-3

```
xxxwww@ubuntu:~/Desktop/Pro$ gcc 07deadlock.c -lpthread
xxxwww@ubuntu:~/Desktop/Pro$ ./a.out
Main: 开始运行，演示死锁现象（程序会永远卡住）
------------------------------------------------
Worker B: 试图获取锁 m2...
Worker A: 试图获取锁 m1...
Worker A: 成功获取锁 m1，正在执行操作 (sleep 1)...
Worker B: 成功获取锁 m2，正在执行操作 (sleep 1)...
Worker A: 试图获取锁 m2...
Worker B: 试图获取锁 m1...
```

线程的同步方式-3


死锁环路图（Deadlock Cycle Diagram）
线程 A
持有（Holds）
锁 1
等待（Waiting For）
等待（Waiting For）
锁 2
持有（Holds）
线程 B
Figures/deadlock_cycle.png

线程的同步方式-3

- 最实用的规避法多把锁时统一加锁顺序。所有线程都先拿m1，再拿m2，不要各写各的。

工作线程 A

```
void* worker_a(void* arg) {
  /* 顺序改规矩：先锁 m1，再 m2 */
  pthread_mutex_lock(&m1);
  sleep(1);
  pthread_mutex_lock(&m2);

  /* 操作共享资源... */

  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  return NULL;
}
```

工作线程 B

```
void* worker_b(void* arg) {
  /* 顺序也要统一：先锁 m1，再 m2 */
  pthread_mutex_lock(&m1);
  sleep(1);
  pthread_mutex_lock(&m2);

  /* 操作共享资源... */

  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  return NULL;
}
```

08deadlock_solved.c

线程的同步方式-3

- 规避原理：两个线程都要先抢m1。一旦A拿到m1，B在想继续往下走时就会被卡在第一道门（阻塞排队等m1）
- 此时由于没人跟A抢m2，A也能顺理成章把两道门都打开完事。成功打破了死锁的环路等待！

```
xxxwww@ubuntu:~/Desktop/Pro$ gcc 08deadlock_solved.c -lpthread
xxxwww@ubuntu:~/Desktop/Pro$ ./a.out
Main: 开始运行，演示解除死锁（统一加锁顺序）
------------------------------------------------
Worker A: 试图获取锁 m1...
Worker A: 成功获取锁 m1，正在执行操作 (sleep 1)...
Worker B: 规避死锁，我也老老实实先试图获取锁 m1...
Worker A: 试图获取锁 m2...
Worker A: 成功获取锁 m2！此时两把锁均已拿到，开始做核心业务。
Worker B: 成功获取锁 m1，正在执行操作 (sleep 1)...
Worker B: 试图获取锁 m2...
Worker B: 成功获取锁 m2！此时两把锁均已拿到，开始做核心业务。
------------------------------------------------
Main: 线程巧妙避开死锁，全部成功执行完毕，安全退出程序！
xxxwww@ubuntu:~/Desktop/Pro$
```

线程的同步方式-4

- 其他同步方式条件变量
- 作用：让线程在某个条件满足时被唤醒，避免轮询（busy-wait）浪费 CPU
- 常用函数：pthread_cond_init，pthread_cond_wait，pthread_cond_signal/broadcast。
- 必须与互斥锁配合使用。

线程的同步方式

- 互斥锁解决什么
  - 谁能进入关键区
  - 谁能修改共享数据
  - 同一时刻只能有一个线程操作
- 条件变量解决什么
  - 条件没满足时谁先等待
  - 条件满足后怎样唤醒等待者
  - 避免线程不停轮询空转

线程相关的函数

- pthread_join
  - 调用 线程.join() 的线程（通常是主线程）会阻塞等待，直到子线程运行结束，才继续往下执行
  - 默认行为：线程默认就是join 的
- pthread_detach
  - 子线程会脱离主线程的控制，在后台独立运行
  - 主线程不再等待子线程，两者各跑各的
  - 主线程退出后，子线程会被操作系统接管，继续运行直到结束

线程相关的函数

- pthread_join示例：主线程等待子线程结束

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

void* worker(void* arg) {
  printf("worker: start\n");
  printf("worker: finish\n");
  return NULL;
}

int main(void) {
  pthread_t tid;
  pthread_create(&tid, NULL, worker, NULL);

  printf("main: before join\n");
  pthread_join(tid, NULL);
  printf("main: after join\n");
  return 0;
}
```

```
(base) chulijie@chulijiedeMacBook-Pro code % ./join
main: before join
worker: start
worker: finish
main: after join
```

09join.c

线程相关的函数

- pthread_detach示例：线程结束后自动回收

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

void* worker(void* arg) {
  printf("worker: running\n");
  sleep(1);
  printf("worker: finish\n");
  return NULL;
}

int main(void) {
  pthread_t tid;
  pthread_create(&tid, NULL, worker, NULL);
  pthread_detach(tid);

  printf("main: detached\n");
  pthread_exit(NULL);
}
```

10detach.c

```
(base) chulijie@chulijiedeMacBook-Pro code % ./detach
main: detached
worker: running
worker: finish
```

相关实例

- 例题1：并发下载模拟
- 问题描述
  - 模拟下载多个耗时不同的文件，对比串行下载和多线程并发下载的总耗时差异。
- 输入与要求
  - 采用结构体硬编码三个任务：A耗时2秒，B耗时1秒，C耗时3秒。
  - 分别执行串行和并行方法，并利用time()获取时间差。
- 输出样例期望
  - 串行总耗时：6(2+1+3)
  - 并发总耗时：3(取决于耗时最长的C)

相关实例

- 例题1：并发下载模拟-解题思路
- 1)用结构体进行参数打包传递设计带有name和cost的任务单元结构体，通过指针传给线程。
- 2)串行vs多线程并发的核心差异
  - 串行：用一个for循环直接调用耗时函数，必须等上一个返回。
  - 并发：在for循环中连发3次pthread_create，把阻塞分散到各线程。随后再开一个for循环负责集中使用pthread_join进行收尾

相关实例

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

struct Task { // 数据任务打包
    char name[16];
    int cost;
};

void* download(void* arg) {
    struct Task* t = (struct Task*)arg;
    sleep(t->cost); // 模拟耗时
    printf("完成下载: %s\n", t->name);
    return NULL;
}
```

11download_sim.c

相关实例

```
int main() {
    struct Task tasks[3] = { {"A", 2}, {"B", 1}, {"C", 3} };

    time_t start = time(NULL);
    for (int i = 0; i < 3; i++) download(&tasks[i]); // 串行跑
    printf("串行总耗时: %ld\n\n", time(NULL) - start);

    start = time(NULL);
    pthread_t tids[3];
    for(int i = 0; i < 3; i++) // 同时发起下载
        pthread_create(&tids[i], NULL, download, &tasks[i]);

    for(int i = 0; i < 3; i++) // 统一回收尸体
        pthread_join(tids[i], NULL);

    printf("并发总耗时: %ld\n", time(NULL) - start);
    return 0;
}
```

```
(base) chulijie@chulijiedeMacBook-Pro code % ./download_sim
完成下载: A
完成下载: B
完成下载: C
串行总耗时: 6

完成下载: B
完成下载: A
完成下载: C
并发总耗时: 3
```

相关实例

- 例题2：大数组并行求和
- 问题描述
  - 给定一个巨大的数组，利用多线程真并行加速累加过程。
- 输入与要求
  - 设定数组长度 N = 10000，元素依次为 0, 1, 2, ..., 9999。开启K = 4 个线程。
  - 要求把计算任务均分给各线程，主线程最后统一汇总各路结果
  - 输出样例期望
  - 最终总和：49995000 (4 路线程最后由主线程融合计算得出)

相关实例

- 例题2：大数组并行求和-解题思路
- 1)划分线程区域
  - 设计带有start，end以及sum的结构体。给每个线程分配它专属的起止索引，互不干扰地操作一块独立的数组内存
- 2)主线程进行管理
  - 创建阶段：算好每一份任务的step，发配任务
  - 回收阶段：通过pthread_join保证该线程任务完成，随后将存在其结构体里的局部sum加上去

相关实例

```c
#include <stdio.h>
#include <pthread.h>
#define N 10000
#define K 4
int arr[N];

struct Range { // 每一个线程的责任田
    int start;
    int end;
    long long sum; // 记录局部结果
};

void* sum_worker(void* arg) {
    struct Range* r = (struct Range*)arg;
    r->sum = 0;
    for (int i = r->start; i < r->end; i++) {
        r->sum += arr[i]; // 仅操作属于自己的那段，不用加锁!
    }
```

12parallel_sum.c

相关实例

```c
int main() {
    for (int i = 0; i < N; i++) arr[i] = i; // 假数据初始化为 0~9999
    pthread_t tids[K];
    struct Range rg[K];
    int step = N / K;
    for (int i = 0; i < K; i++) { // 包工头发包
        rg[i].start = i * step;
        rg[i].end = (i == K-1) ? N : (i+1)*step;
        pthread_create(&tids[i], NULL, sum_worker, &rg[i]);
    }
    long long total = 0;
    for (int i = 0; i < K; i++) { // 包工头查收计算结果
        pthread_join(tids[i], NULL);
        total += rg[i].sum;
    }
    printf("最后总和: %lld\n", total);
    return 0;
}
```

```
(base) chulijie@chulijiedeMacBook-Pro code % gcc -o parallel_sum parallel_sum.c
(base) chulijie@chulijiedeMacBook-Pro code % ./parallel_sum
最后总和: 49995000
```

谢谢！

中国科学技术大学
University of Science and Technology of China