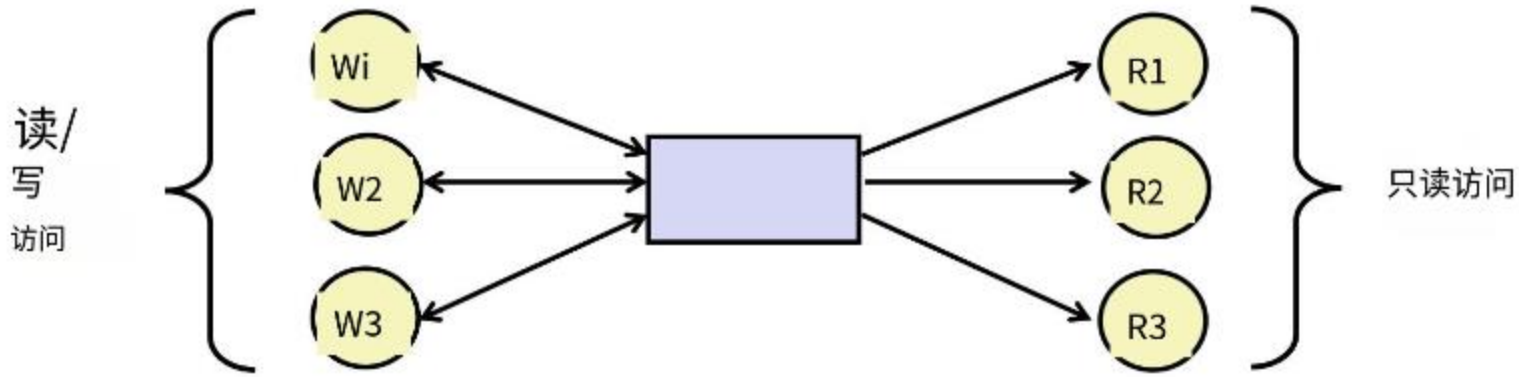
SYNC Basic

徐品原

回顾:生产者-消费者问题

- 常用同步模式:
  - 生产者等待空槽，在缓冲区中插入项，并通知消费者
  - 消费者等待项目，将其从缓冲区中删除，并通知生产者
- 例子
  - 多媒体处理:
    - 生产者创建视频帧，消费者呈现事件驱动的图形用户界面
    - 生产者检测鼠标点击、鼠标移动和键盘敲击，并在缓冲区中插入相应的事件
    - 消费者从缓冲区中检索事件并绘制显示

Readers-Writers问题



- 问题陈述:
  - 读取线程只读取对象
  - 写线程修改对象(读/写访问)
  - 编写器必须对对象具有独占访问权
  - 无限数量的阅读器可以访问该对象
- 在实际系统中经常发生，例如，
  - 网上机票预订系统
  - 多线程缓存Web代理

读者-作者的变体

- 第一读者-作者问题(偏爱读者)
  - 除非写对象已经被授予使用对象的权限，否则不应该让读对象等待。
  - 在等待的写入器之后到达的读取器优先于该写入器。
- 第二读者-作者问题(偏爱作者)
  - 一旦写入器准备好写入，它就会尽快执行写入
  - 在写入器之后到达的读者必须等待，即使写入器也在等待。
- 在botcase中可能会出现饥饿(线程无限期等待)。

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

Writers:

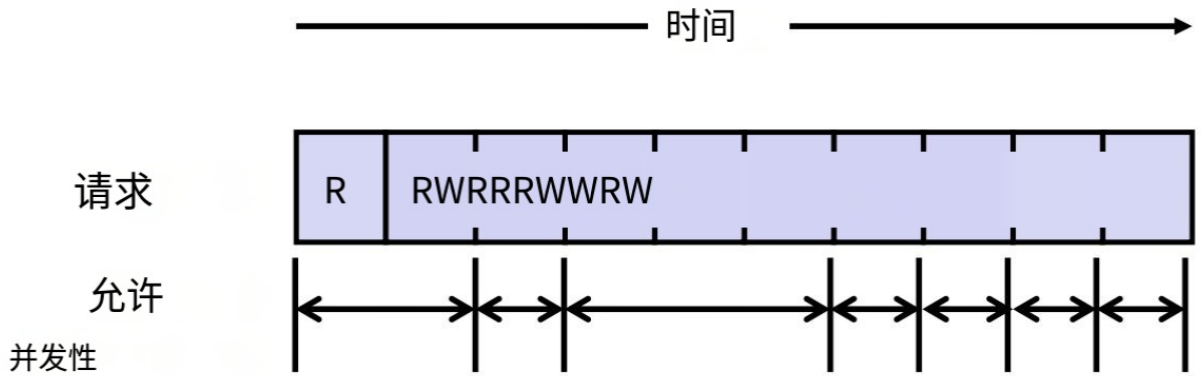
```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

```
int readcnt, writecnt;       // Initially 0
sem_t rmutex, wmutex, r, w; // Initially 1
void reader(void)
{
  while (1) {
    P(&r);
    P(&rmutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&rmutex);
    V(&r)

    /* Reading happens here */

    P(&rmutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&rmutex);
  }
}
```

```
void writer(void)
{
  while (1) {
    P(&wmutex);
    writecnt++;
    if (writecnt == 1)
        P(&r);
    V(&wmutex);

    P(&w);
    /* Writing here */
    V(&w);

    P(&wmutex);
    writecnt--;
    if (writecnt == 0);
        V(&r);
    V(&wmutex);
  }
}
```

FIFO管理读写器



- 的想法
  - 读和写请求被插入FIFO
  - 请求被处理为从FIFO移除
    - 如果当前空闲或正在处理读取，则允许继续读取
    - 写只允许在空闲时进行
  - 请求完成后通知控制器
- 公平
  - 保证每个请求最终都得到处理

- **Full code in rwqueue.{h,c}**

```
/* Queue data structure */
typedef struct {
    sem_t mutex;  // Mutual exclusion
    int reading_count;    // Number of active readers
    int writing_count;    // Number of active writers
    // FIFO queue implemented as linked list with tail
    rw_token_t *head;
    rw_token_t *tail;
} rw_queue_t;
```

```
/* Represents individual thread's position in queue */
typedef struct TOK {
    bool is_reader;
    sem_t enable;        // Enables access
    struct TOK *next;    // Allows chaining as linked list
} rw_token_t;
```

- In rwqueue-test.c

```c
/* Get write access to data and write */
void iwriter(int *buf, int v)
{
    rw_token_t tok;
    rw_queue_request_write(&q, &tok);
    /* Critical section */
    *buf = v;
    /* End of Critical Section  */
    rw_queue_release(&q);
}
```

```c
/* Get read access to data and read */
int ireader(int *buf)
{
    rw_token_t tok;
    rw_queue_request_read(&q, &tok);
    /* Critical section */
    int v = *buf;
    /* End of Critical section */
    rw_queue_release(&q);
    return v;
}
```

关键概念:线程安全从线程中调用的函数必须是线程安全的

- 函数是线程安全的，如果它在多个并发线程中重复调用时总是产生正确的结果。

- 线程不安全函数的类:
  - 类1:不保护共享变量的函数
  - 类2:跨多个调用保持状态的函数
  - 类3:返回指向静态变量的指针的函数
  - 第4类:调用线程不安全函数的函数

线程不安全函数(第1类)

- 未能保护共享变量
  - 修正:使用P和V信号量操作
  - 例如:goodcnt.c
  - 问题:同步操作将减慢代码速度

线程不安全函数(第2类)

- 依靠跨多个函数调用的持久状态
  - 示例:依赖于静态状态的随机数生成器

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

线程安全随机数生成器

- 传递状态作为参数的一部分
  - 因此，消除了静态

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- 结果:使用rand_r的程序员必须维护种子

线程不安全函数(第3类)

- 返回指向静态变量的指针
- 解决1。重写函数，使调用者传递变量的地址来存储结果
  - 要求更改调用方和被调用方
- 修复2。Lock-and-copy
  - 需要在调用方中进行简单的更改(而在被调用方中不需要更改)。

```
/* Convert integer to string */
char *itoa(int x)
{
     static char buf[11];
     sprintf(buf, "%d", x);
     return buf;
}
```

```
char *lc_itoa(int x, char *dest)
{
     P(&mutex);
     strcpy(dest, itoa(x));
     V(&mutex);
     return dest;
}
```

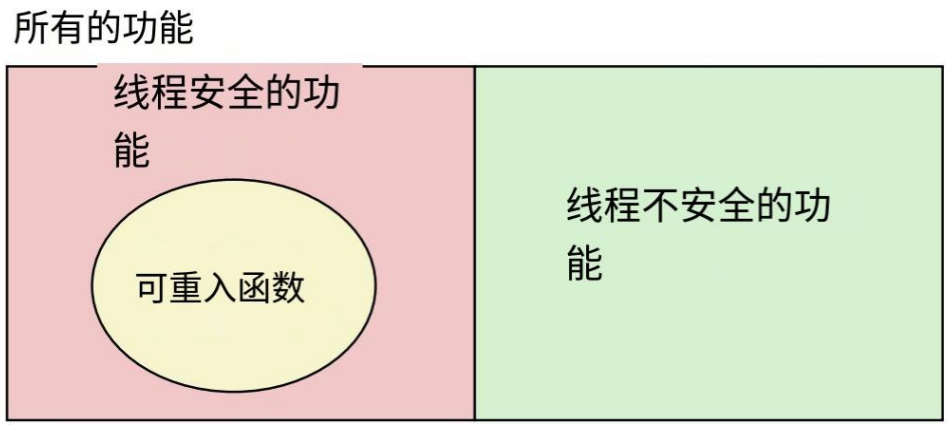
警告:像gethostbyname这样的函数需要深度拷贝。使用可重入gethostbyname_rversion代替。

线程不安全函数(第4类)

- 调用线程不安全的函数
  - 调用一个线程不安全的函数会使调用它的整个函数都是线程不安全的
  - 修正:修改函数，使其只调用线程安全的函数 ☺

可重入函数

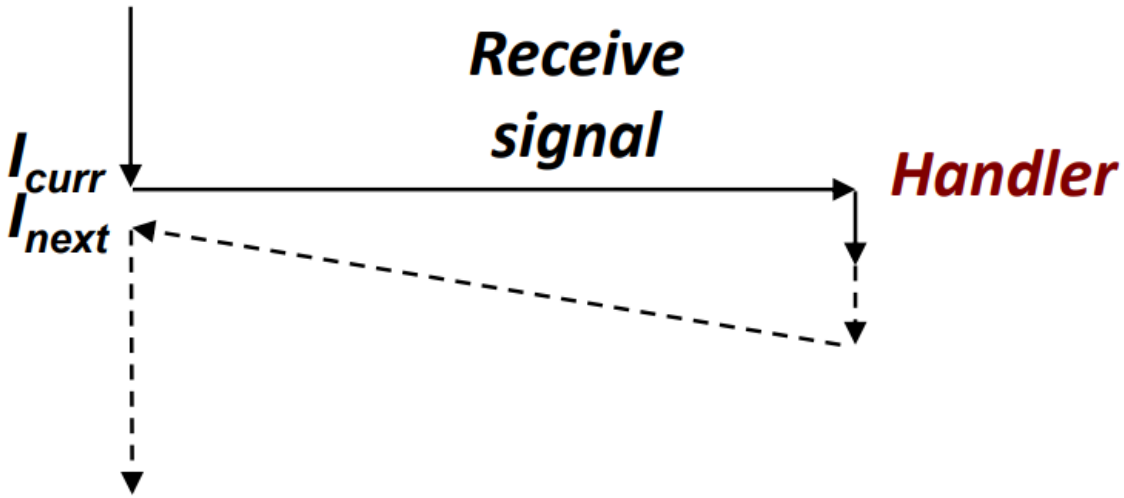
- 如果一个函数在被多个线程调用时没有访问共享变量，那么它就是可重入的。
  - 线程安全函数的重要子集
    - 不需要同步操作
    - 使Class 2函数线程安全的唯一方法是使其可重入(例如，rand_r)。



线程安全的库函数

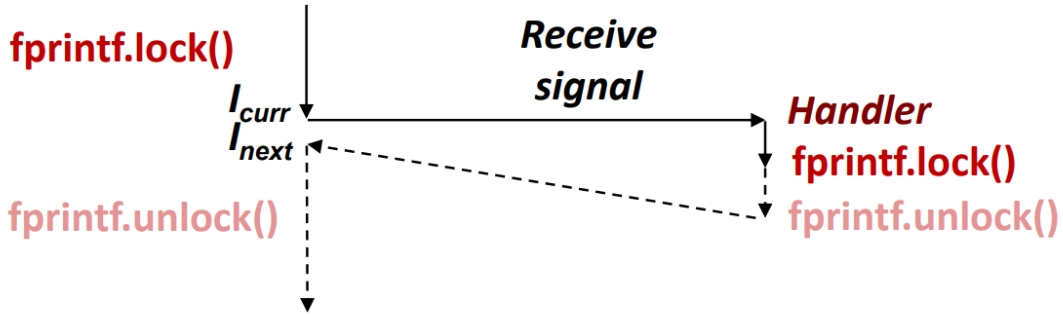
- 标准C库中的所有函数(在K&R文本的后面)都是线程安全的
  - 示例:malloc, free, printf, scanf大多数Unix系统调用都是线程安全的，除了一些例外:

Thread-unsafe function	Class	Reentrant version
asctime	3	asctime_r
ctime	3	ctime_r
gethostbyaddr	3	gethostbyaddr_r
gethostbyname	3	gethostbyname_r
inet_ntoa	3	(none)
localtime	3	localtime_r
rand	2	rand_r



- 行动
  - 信号可以发生在程序执行的任何时刻
    - 除非信号被阻断
  - 信号处理程序在同一线程内运行
  - 必须运行到完成，然后返回到常规程序执行

- 为了线程安全，许多库函数使用了锁与复制
  - 因为它们有隐藏状态
  - malloc
    - 自由列表
  - 打印，打印，放入
    - 因此，来自多个线程的输出不会交错
  - sprintf
    - 不是正式的异步信号安全，但似乎是可以的
- 对于不使用这些库函数的处理程序



- 如果:
  - 当库函数持有锁时收到信号处理程序调用相同(或相关)库函数死锁!
-
  - 信号处理程序在被锁定之前不能继续
  - 主程序不能继续，直到处理程序完成关键点
-
  - 线程采用对称并发
  - 信号处理是不对称的

线程的总结

- 线程为编写并发程序提供了另一种机制
- 线程越来越受欢迎
  - 比工艺便宜一些
  - 易于在线程之间共享数据
- 然而，共享的便利性是有代价的:
  - 容易引入细微的同步错误
  - 小心踩线!