

进

程

间

通

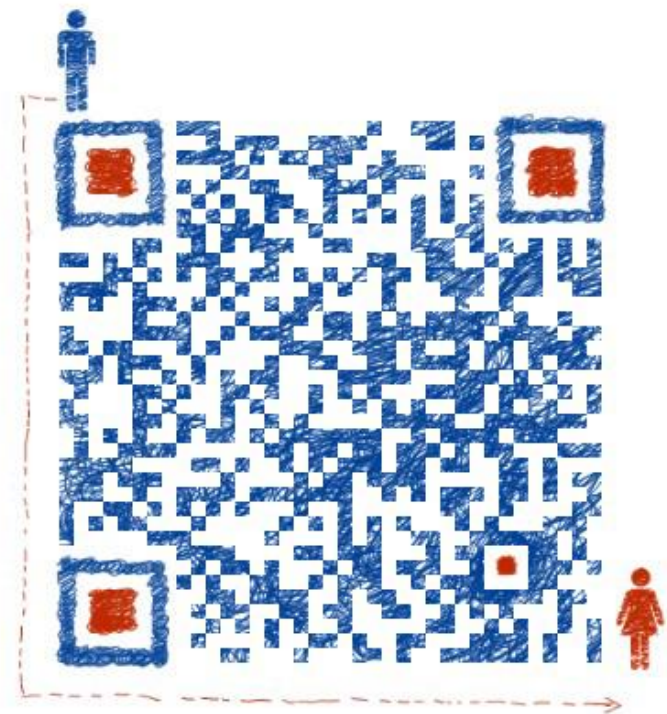
信

—Linux

无线传感器网项目实战

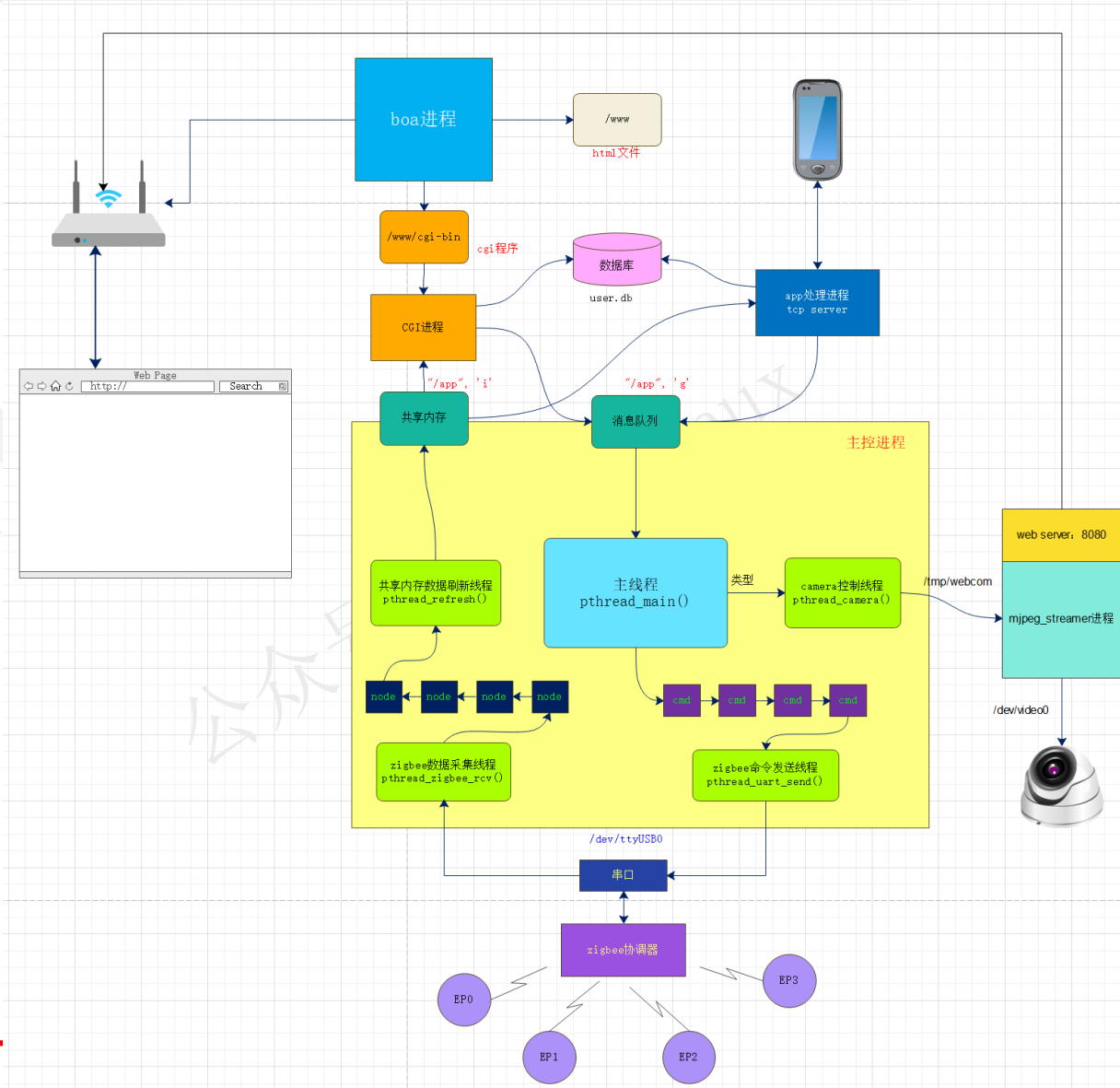


公众号:一口Linux



彭老师个人微信号

本项目使用到的进程间通信



关注公众号：—

01

进程间通信

进程间通信概述



AT&T

UNIX平台进程通信方式

- 早期进程间通信方式
 - 管道、有名管道和信号
- AT&T的贝尔实验室
 - 对Unix早期的进程间通信进行了改进和扩充，形成了“system V IPC”，其通信进程主要局限在单个计算机内
- BSD(加州大学伯克利分校的伯克利软件发布中心)
 - 形成了基于套接字(socket)的计算机之间的进程间通信机制
- Linux继承了上述所有的通信方式



Linux下进程间通信概述

- 常用的进程间通信方式

- 传统的进程间通信方式

- 无名管道(pipe)、有名管道(fifo)和信号(signal)

- System V IPC对象

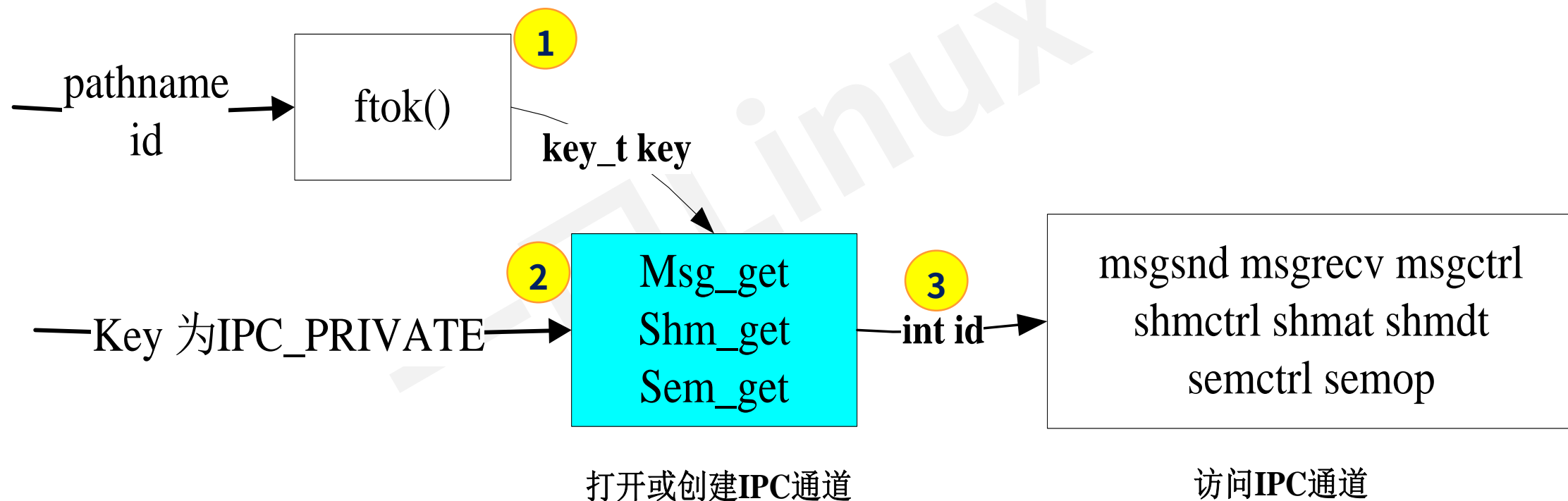
- 共享内存(share memory)、消息队列(message queue)和信号灯(semaphore)

- BSD

- 套接字(socket)

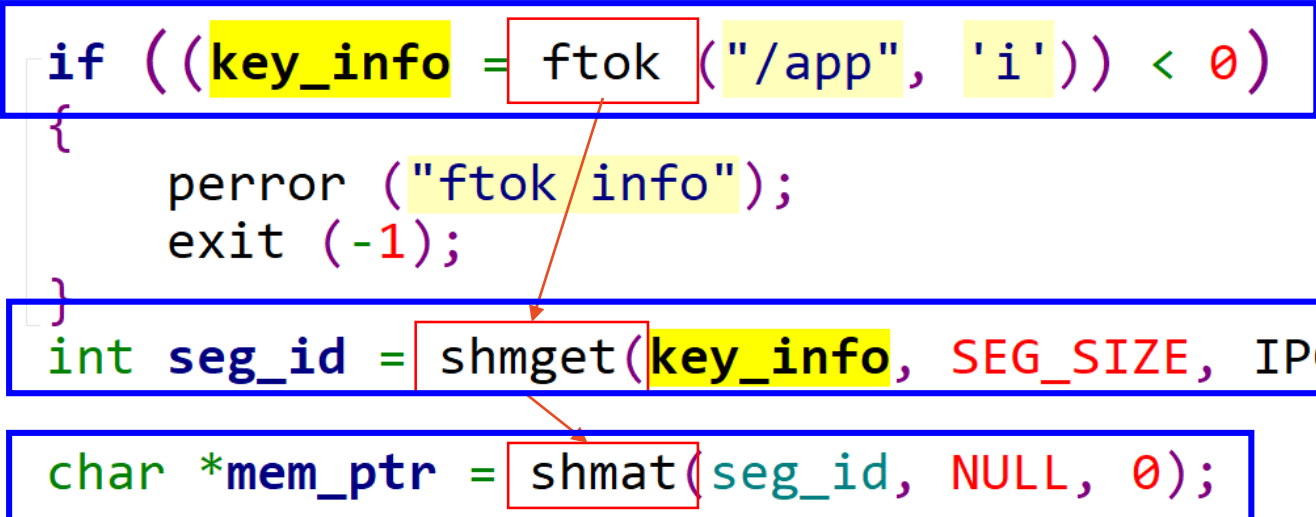
IPC对象

- IPC(Inter-Process Communication)进程间通信，提供了各种进程间通信的方法



举例-共享内存

```
43: key_t key_info;  
44:  
45: if ((key_info = ftok("/app", 'i')) < 0)  
46: {  
47:     perror("ftok info");  
48:     exit(-1);  
49: }  
50: int seg_id = shmget(key_info, SEG_SIZE, IPC_CREAT | 0777);  
51:  
52: | char *mem_ptr = shmat(seg_id, NULL, 0);  
53:
```



ipcs、ipcrm

1.ipcs命令用于查看系统中的IPC对象

1.ipcs -m 共享内存

2.ipcs -s 信号量

3.ipcs -q 消息队列

2.ipcrm命令用于删除系统中的IPC对象

ipcrm -m *id*

```
peng@ubuntu:~/work/test/wait$ ipcs
----- Message Queues -----
key          msqid        owner         perms         used-bytes   messages
----- Shared Memory Segments -----
key          shmid        owner         perms         bytes        nattch       status
0x00000000   294912       peng          600           524288       2            dest
0x00000000   393217       peng          600           524288       2            dest
0x00000000   1507330      peng          600           524288       2            dest
0x00000000   524291       peng          600           524288       2            dest
0x00000000   786436       peng          600           524288       2            dest
0x00000000   884741       peng          600           524288       2            dest
0x00000000   983046       peng          600           524288       2            dest
0x00000000   1409031      peng          600           524288       2            dest
0x00000000   1048584      peng          600           16777216     2            dest
0x00000000   1277961      peng          600           524288       2            dest
0x00000000   1310730      peng          600           134217728    2            dest
0x00000000   1605643      peng          600           524288       2            dest
0x00000000   1638412      peng          600           268435456    2            dest
0x00000000   1671181      peng          600           1073741824   2            dest
0x6901000f   3276814      peng          777           100          0            dest
0x00000000   2261007      peng          600           524288       2            dest
0x00000000   3145745      peng          600           4194304       2            dest
----- Semaphore Arrays -----
key          semid        owner         perms         nsems
```

创建的IPC对象如果不删除的话会一直保留在系统中

02

共享内存

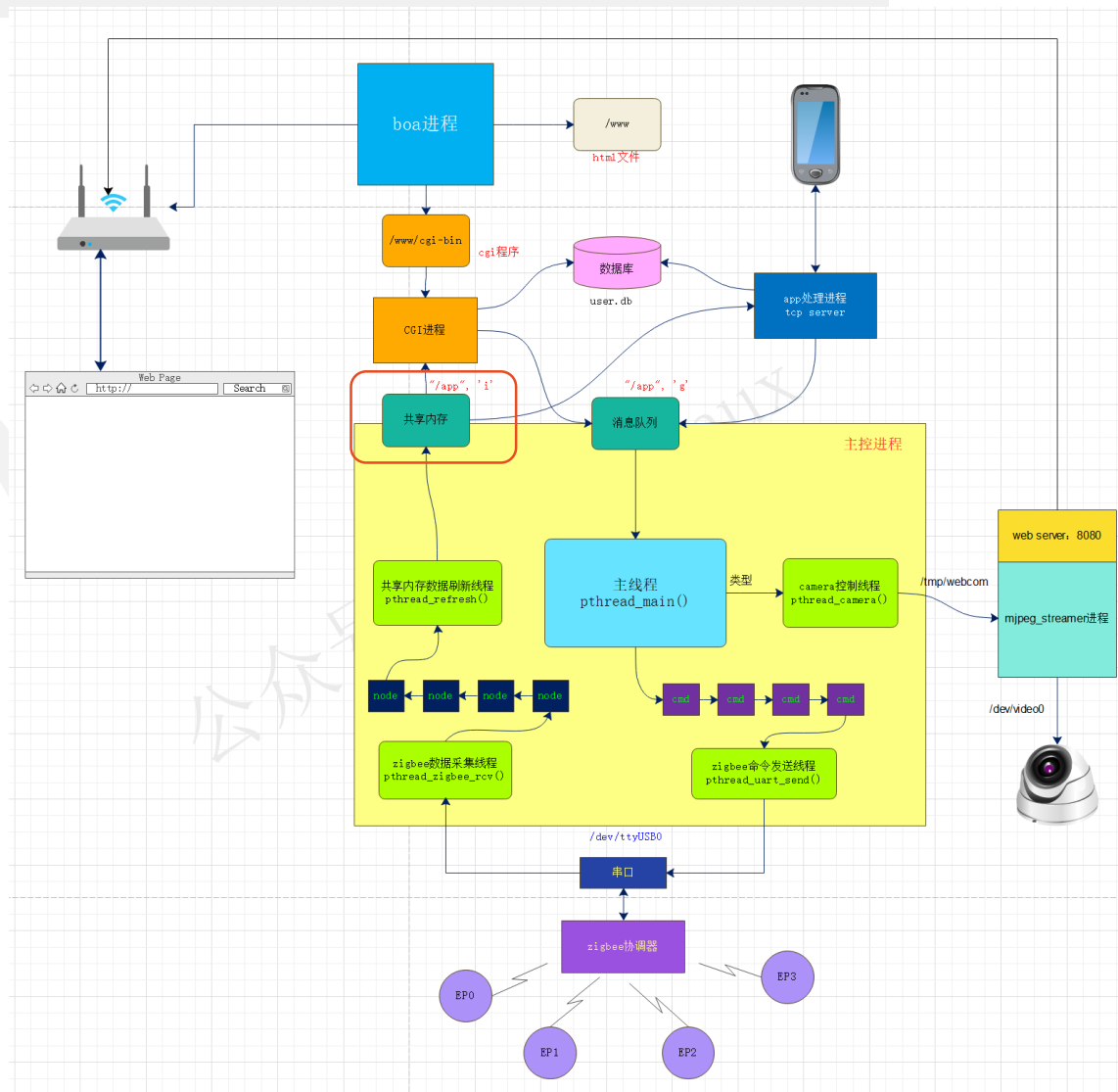
共享内存

- 共享内存是一种**最为高效**的进程间通信方式，进程可以直接读写内存，而不需要任何数据的拷贝
- 为了在多个进程间交换信息，**内核**专门留出了一块内存区，可以由需要访问的进程将其**映射到自己的私有地址空间**
- 进程就可以**直接读写**这一内存区而不需要进行数据的拷贝，从而大大提高的效率。
- 由于多个进程共享一段内存，因此也需要依靠某种同步机制，如**互斥锁**和**信号量**等

共享内存存在该项目中使用

- 从串口读取的zigbee网络环境数据要发送给web页面或者APP，必须满足：

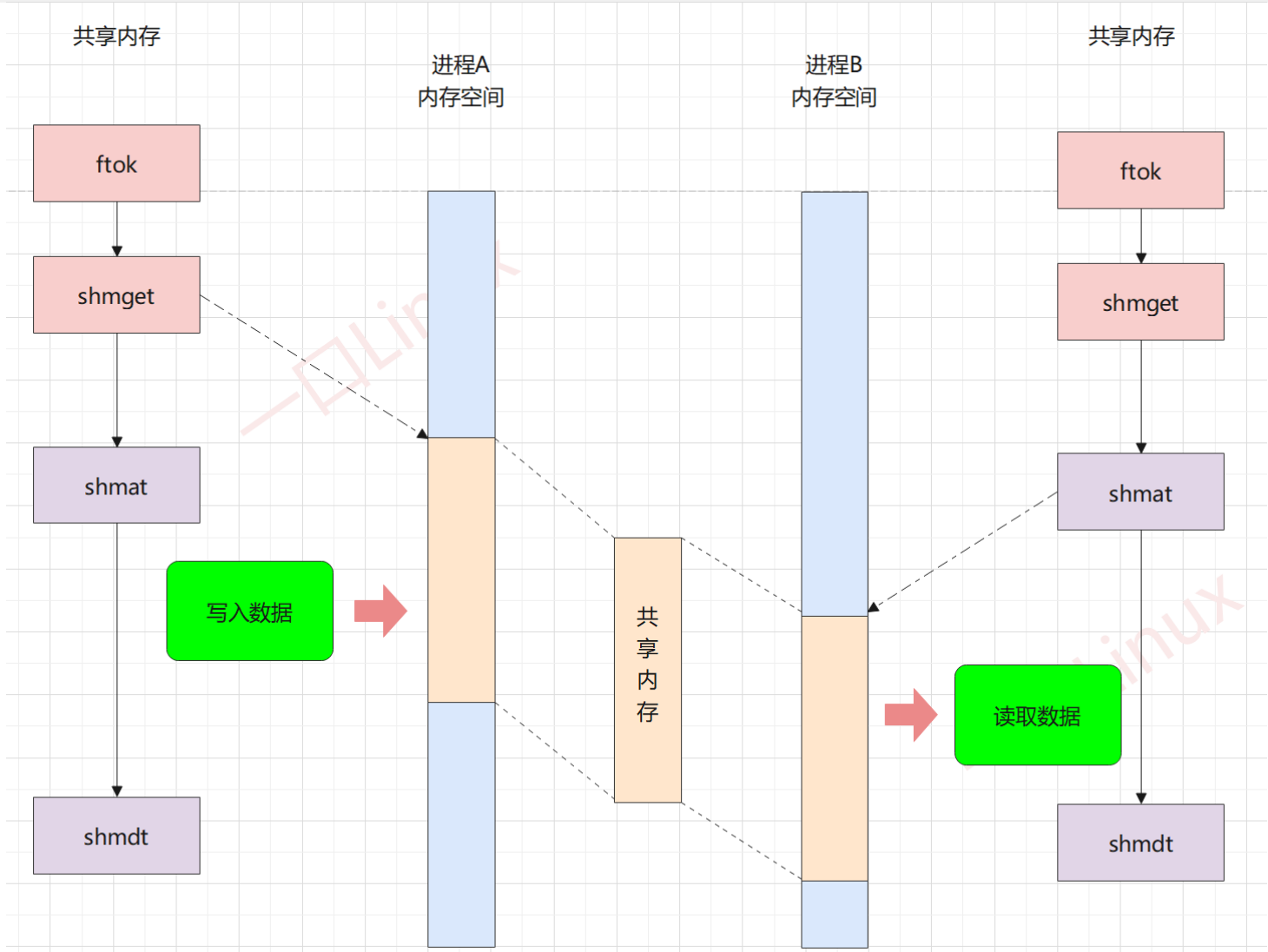
- 1.所有的其他进程都可以定时从“某块内存”读取数据
- 2.有新的数据更新时，可以很方便的将数据写入到“某块内存”
- 3.读取的数据不需要清除原有数据，写入的数据要更新到这块内存



共享内存实现

- 共享内存的使用包括如下步骤：
 - 1.创建/打开共享内存
 - 2.映射共享内存，即把指定的共享内存映射到进程的地址空间用于访问
 - 3.撤销共享内存映射
 - 4.删除共享内存对象

共享内存函数调用流程



共享内存函数shmget

所需头文件 `#include <sys/types.h>`
 `#include <sys/ipc.h>`
 `#include <sys/shm.h>`

函数原型 `int shmget(key_t key, int size, int shmflg);`

函数参数 **key:** IPC_PRIVATE 或 ftok的返回值

size: 共享内存区大小

shmflg: 同open函数的权限位，也可以用8进制表示法

函数返回值 成功：共享内存段标识符

 出错：-1

共享内存函数shmat

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h></pre>
函数原型	<pre>void *shmat(int shmid, const void *shmaddr, int shmflg);</pre>
函数参数	<p>shmid: 要映射的共享内存区标识符</p> <p>shmaddr: 将共享内存映射到指定地址(若为NULL, 则表示由系统自动完成映射)</p> <p>shmflg : SHM_RDONLY: 共享内存只读</p> <p> 默认0: 共享内存可读写</p>
函数返回值	<p>成功: 映射后的地址</p> <p>出错: -1</p>

共享内存函数shmdt

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h></pre>
函数原型	<pre>int shmdt(const void *shmaddr);</pre>
函数参数	shmaddr: 共享内存映射后的地址
函数返回值	成功: 0
	出错: -1

共享内存函数shmctl

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h></pre>
函数原型	<pre>int shmctl(int shmid, int cmd, struct shmid_ds *buf);</pre>
函数参数	<p>shmid: 要操作的共享内存标识符</p> <p>cmd: IPC_STAT (获取对象属性) IPC_SET (设置对象属性) IPC_RMID (删除对象)</p> <p>buf: 指定IPC_STAT/IPC_SET时用以保存/设置属性</p>
函数返回值	<p>成功: 0</p> <p>出错: -1</p>

举例

ipcs -m

shm-client.c

```
main()
{
    key_t key_info;

    if ((key_info = ftok ("/app", 'i')) < 0)
    {
        perror ("ftok info");
        exit (-1);
    }
    int seg_id = shmget(key_info, SEG_SIZE, 0777);

    char *mem_ptr = shmat(seg_id, NULL, 0);

    printf("The time, direct from memory: ..%s", mem_ptr);
    shmdt(mem_ptr);
}
```

shm-server.c

```
main()
{
    long now;
    int n;
    key_t key_info;

    if ((key_info = ftok ("/app", 'i')) < 0)
    {
        perror ("ftok info");
        exit (-1);
    }

    int seg_id = shmget(key_info, SEG_SIZE, IPC_CREAT | 0777);

    char *mem_ptr = shmat(seg_id, NULL, 0);

    for (n = 0; n < 60; n++)
    {
        time(&now);          /* get the time */
        strcpy(mem_ptr, ctime(&now)); /* write to mem */
        sleep(1);             /* wait a sec */
    }
    shmctl(seg_id, IPC_RMID, NULL);
}
```

物联网实训项目所有资料\3.课程代码讲解实例\3.linux系统编程\3.process\ shm

关注公众号：一口Linux

03

消息队列

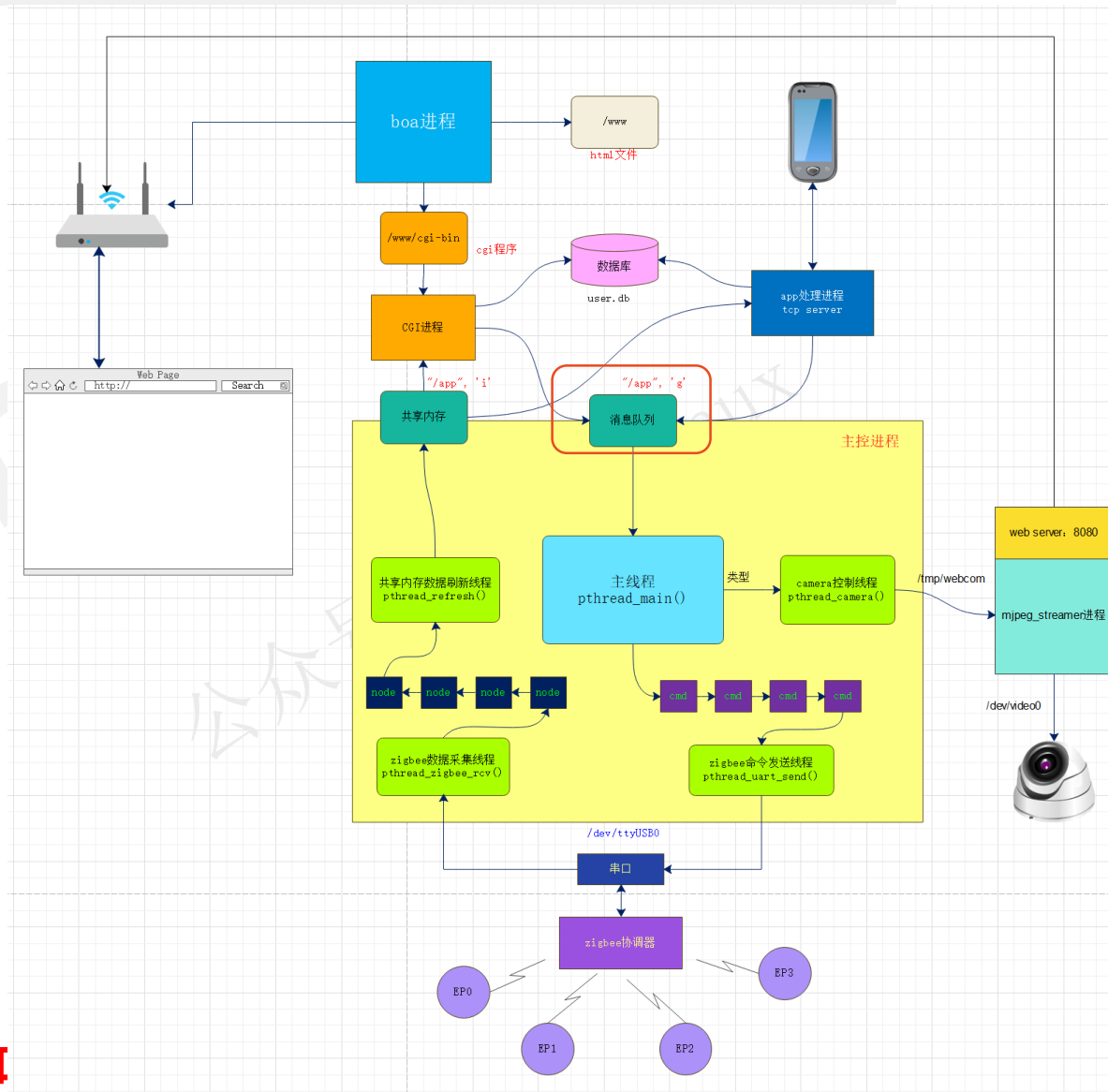
消息队列基本概念

- 消息队列是IPC对象的一种
- 消息队列由消息队列ID来唯一标识
- 消息队列就是一个消息的列表。用户可以在消息队列中添加消息、读取消息等。
- 消息队列可以按照类型来发送/接收消息

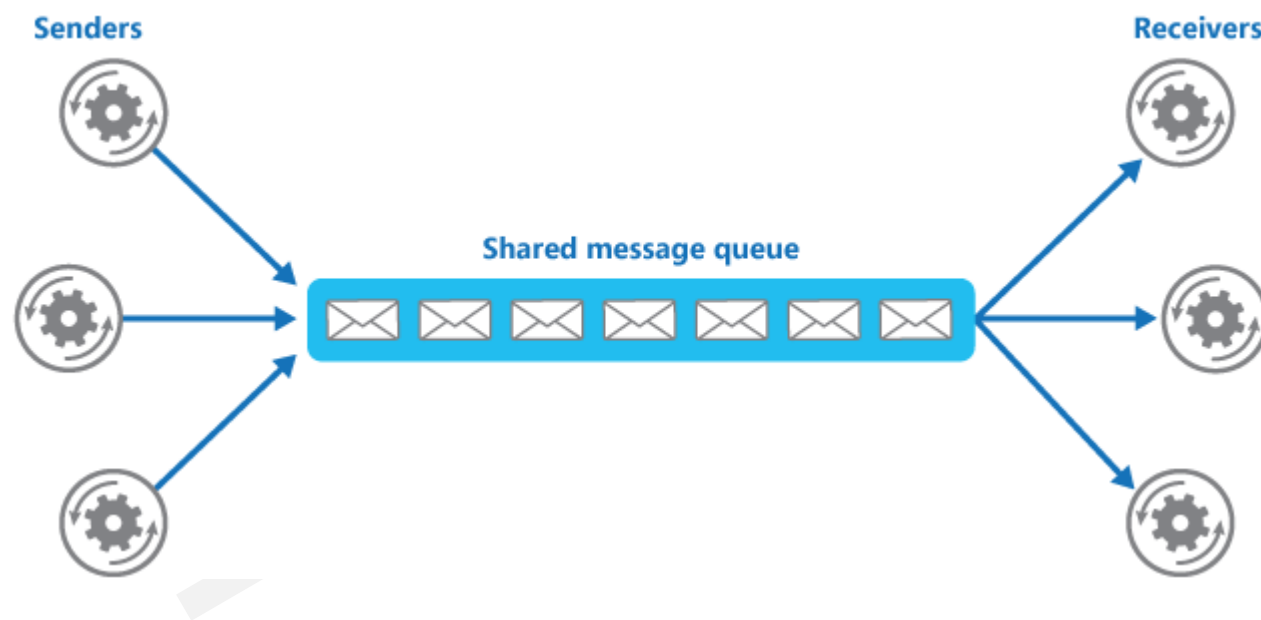
消息队列在该项目中使用

- 从web页面或者APP下发的命令，最终要通过串口发送给zigbee网络的终端节点，以控制各种外设，需要考虑以下问题：

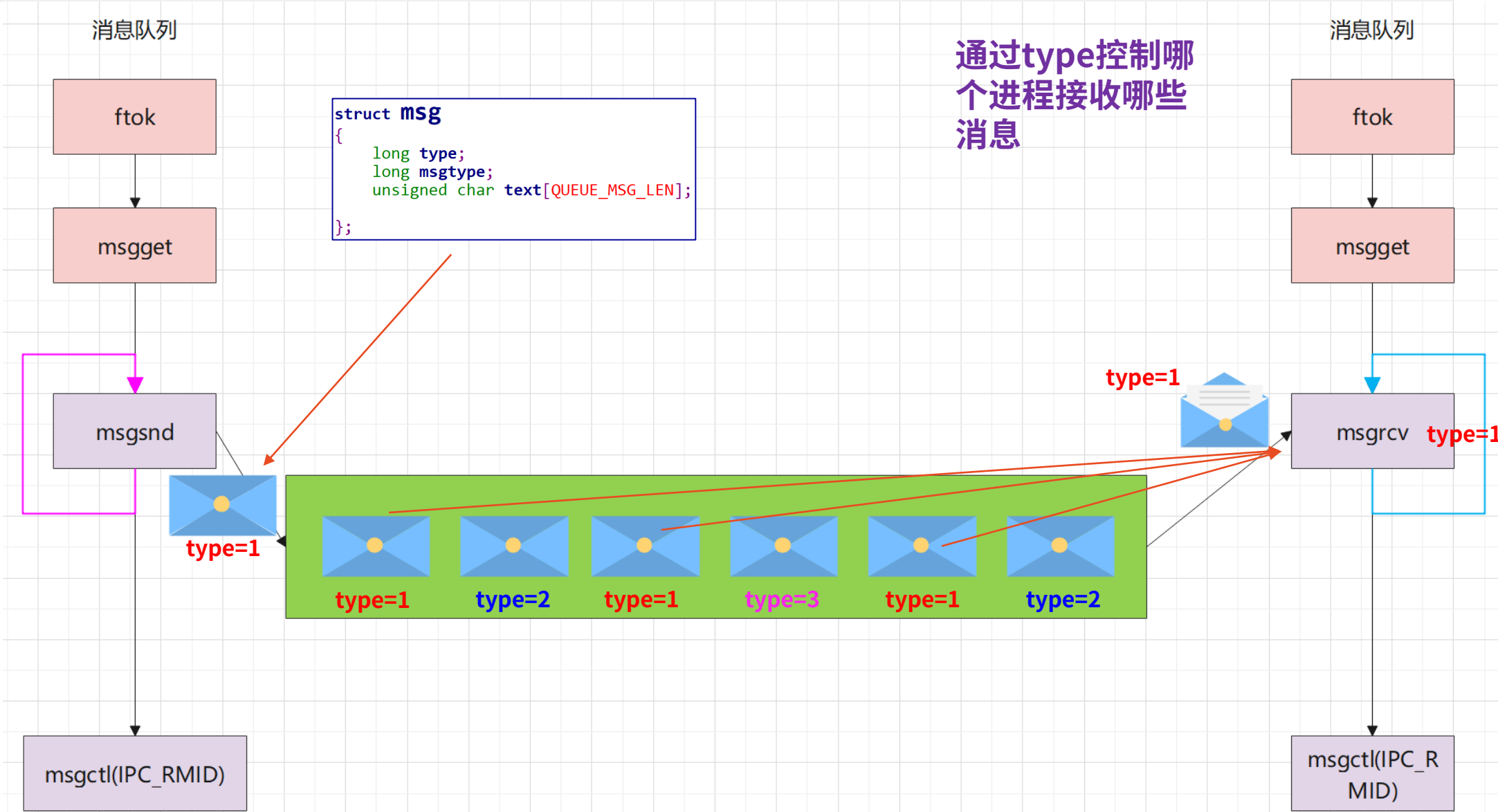
- 1. 所有的命令不能丢失(连续点击页面按钮)
- 2. 命令需要按照一定顺序排列
- 3. zigbee网络可能延时相比较有点大，不能发送太快



消息队列模型



消息队列函数调用流程



消息队列函数-msgget

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h></pre>
函数原型	<pre>int msgget(key_t key, int flag);</pre>
函数参数	key: 和消息队列关联的key值
	flag: 消息队列的访问权限
函数返回值	成功: 消息队列ID
	出错: -1

消息队列函数-msgsnd

所需头文件

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

函数原型

```
int msgsnd(int msqid, const void *msgp, size_t size,
int flag);
```

函数参数

msqid: 消息队列的ID

msgp: 指向消息的指针。常用消息结构msgbuf如下:

```
struct msgbuf{
    long mtype;    //消息类型
    char mtext[N]; //消息正文
```

size: 发送的消息正文的字节数

flag: IPC_NOWAIT 消息没有发送完成函数也会立即返回。

0: 直到发送完成函数才返回

函数返回值

成功: 0

出错: -1

消息队列函数-msgrcv

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h></pre>
函数原型	<pre>int msgrcv(int msgid, void* msgp, size_t size, long msgtype, int flag);</pre>
函数参数	msgid: 消息队列的ID
	msgp: 接收消息的缓冲区
	size: 要接收的消息的字节数
	msgtype: 0: 接收消息队列中第一个消息。 大于0: 接收消息队列中第一个类型为msgtyp的消息。 小于0: 接收消息队列中类型值不小于msgtyp的绝对值且类型值又最小的消息。
	flag: 0: 若无消息函数会一直阻塞 IPC_NOWAIT: 若没有消息, 进程会立即返回ENOMSG。
函数返回值	成功: 接收到的消息的长度 出错: -1

消息队列函数-msgctl

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/msg.h></pre>
函数原型	<pre>int msgctl (int msgqid, int cmd, struct msqid_ds *buf);</pre>
函数参数	<p>msqid: 消息队列的队列ID</p> <p>cmd: IPC_STAT: 读取消息队列的属性, 并将其保存在buf指向的缓冲区中。</p> <p>IPC_SET: 设置消息队列的属性。这个值取自buf参数。</p> <p>IPC_RMID: 从系统中删除消息队列。</p> <p>buf: 消息队列缓冲区</p>
函数返回值	<p>成功: 0</p> <p>出错: -1</p>

实例

- 物联网实训项目所有资料\3.课程代码讲解实例\3.linux系统编程\3.process\msgq

一口Linux

04

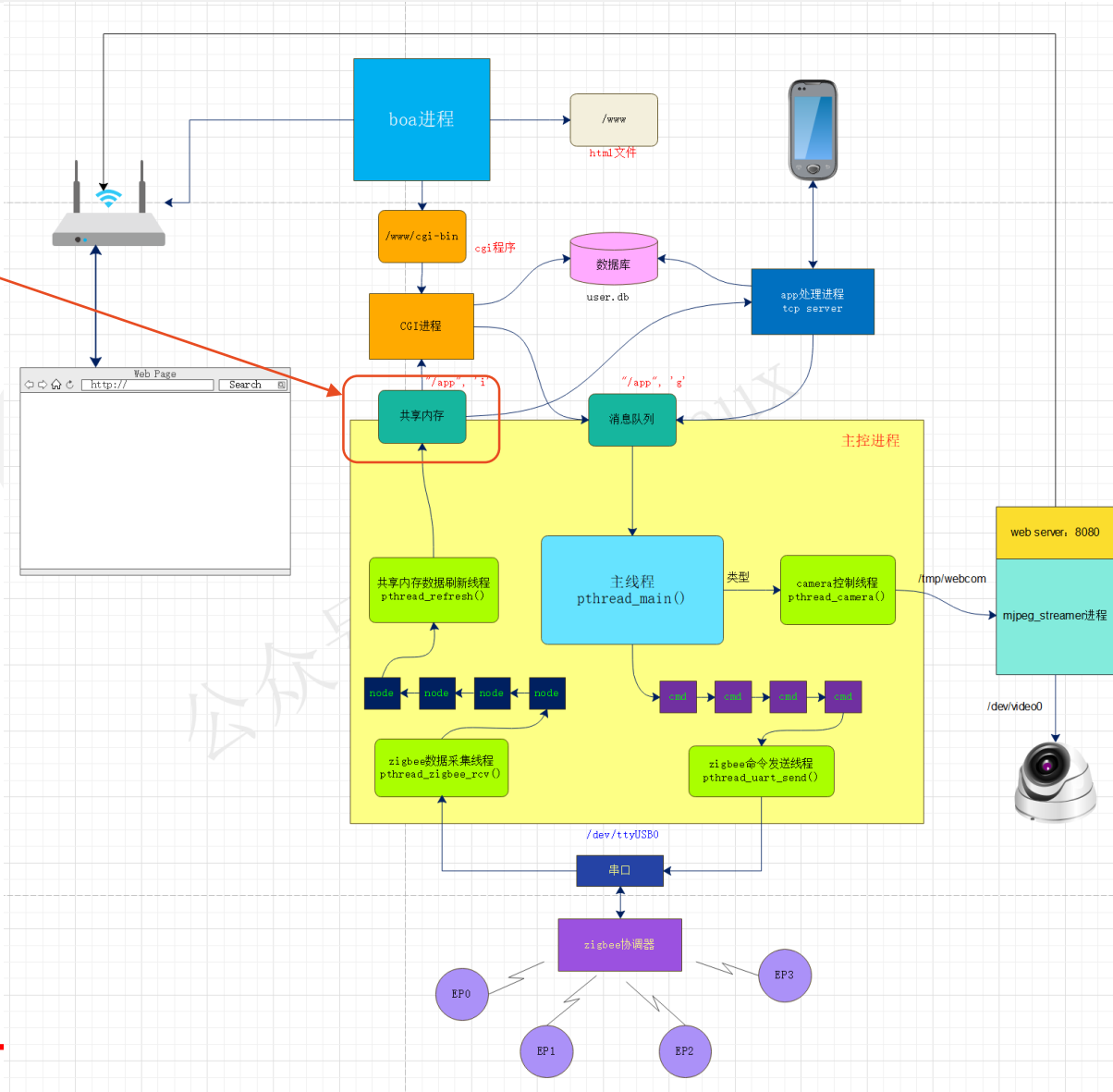
信号灯

临界资源

- 一次只允许一个进程使用的资源称为临界资源；
 - 临界资源并不全是硬件或是软件，而是两者都能作为临界资源。
 - 比如硬件的有：
 - 打印机、磁带机等；
 - 软件有：
 - 消息缓冲队列、变量、数组、缓冲区等；
- 临界区（critical region）
 - 访问共享变量的程序代码段称为临界区，也称为临界段（critical section）；
- 进程互斥
 - 两个或两个以上的进程不能同时进入关于同一组共享变量的临界区，否可能会发生与时间有关的错误，这种现象称为进程互斥；

本项目中的临界资源

- 共享内存需要互斥访问
 - 有进程在访问时，其他进程不能访问
 - 信号量初始值为1



关注公众号：—

信号灯

- 信号灯(semaphore), 也叫信号量。它是不同进程间或一个给定进程内部不同线程间同步的机制。
- 信号灯种类:
 - posix有名信号灯(可用于线程、进程同步)
 - posix基于内存的信号灯(无名信号灯)
 - System V信号灯(IPC对象)

本篇主要讲System V 信号灯



System V 信号灯

- **System V**的信号灯是一个或者多个信号灯的一个集合。其中的每一个都是单独的计数信号灯。
- 而**Posix**信号灯指的是单个计数信号灯
- **System V** 信号灯由内核维护
-

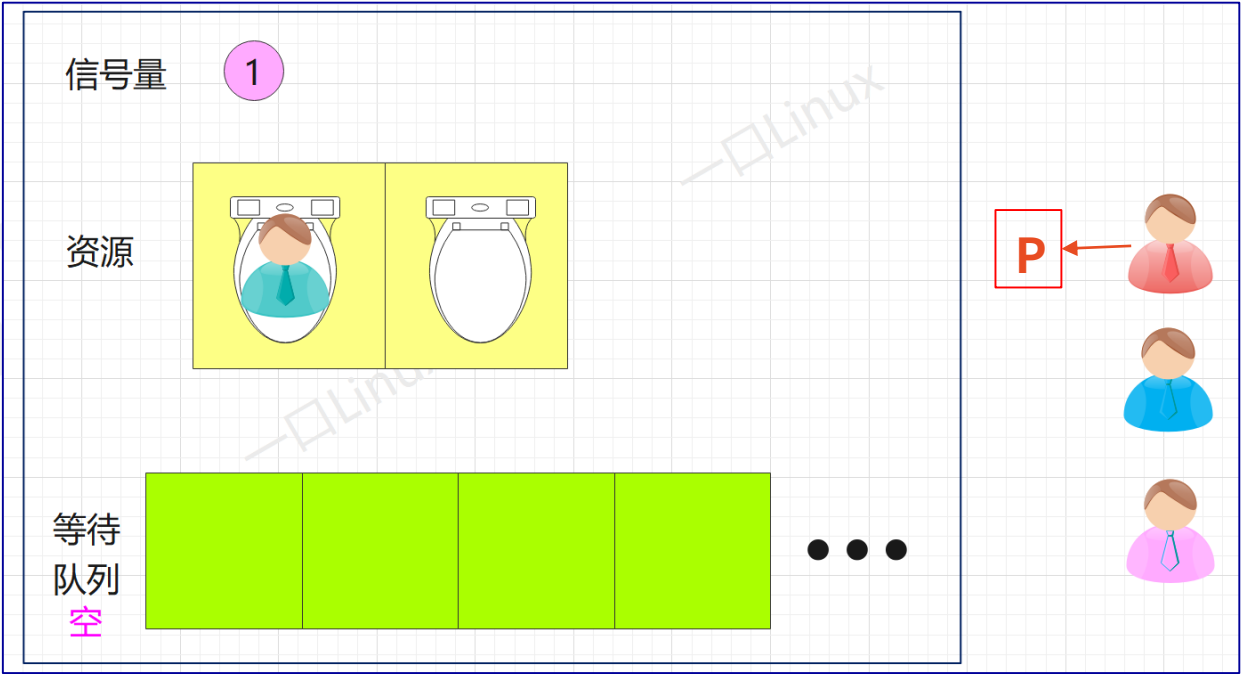
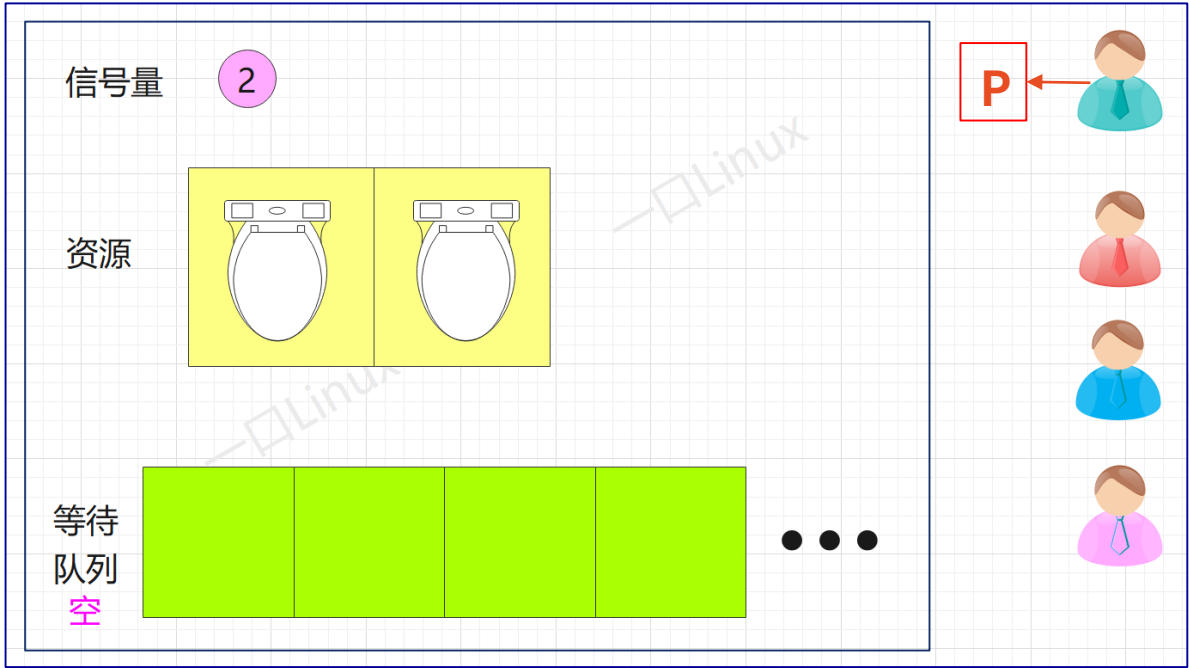
二值信号灯、计数信号灯

- 二值信号灯：
 - 值为0或1。与互斥锁类似，资源可用时值为1，不可用时值为0。
- 计数信号灯：
 - 值在0到n之间。用来统计资源，其值代表可用资源数

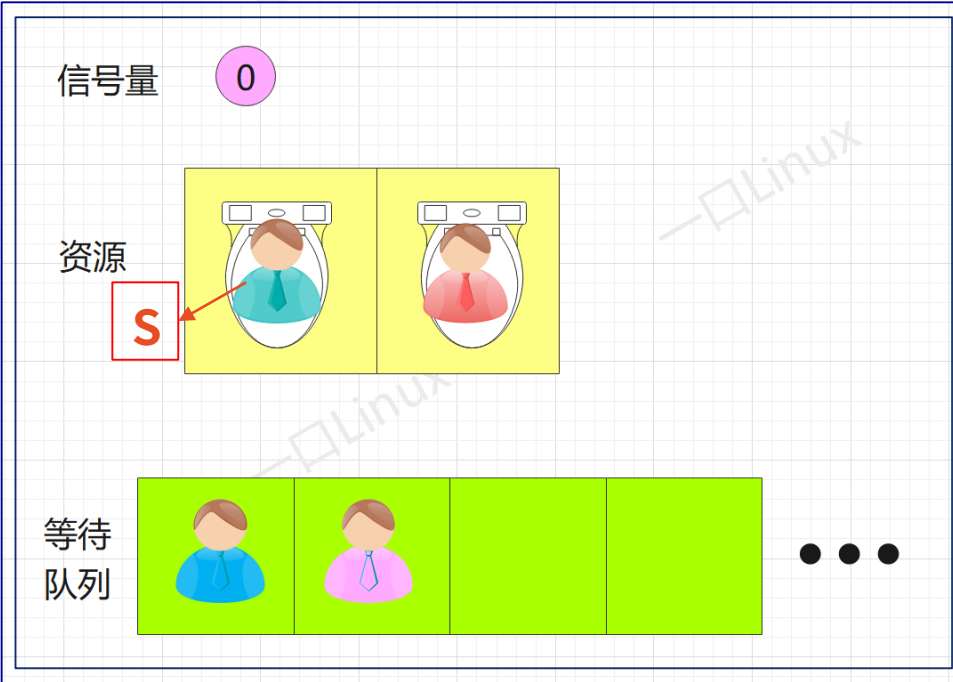
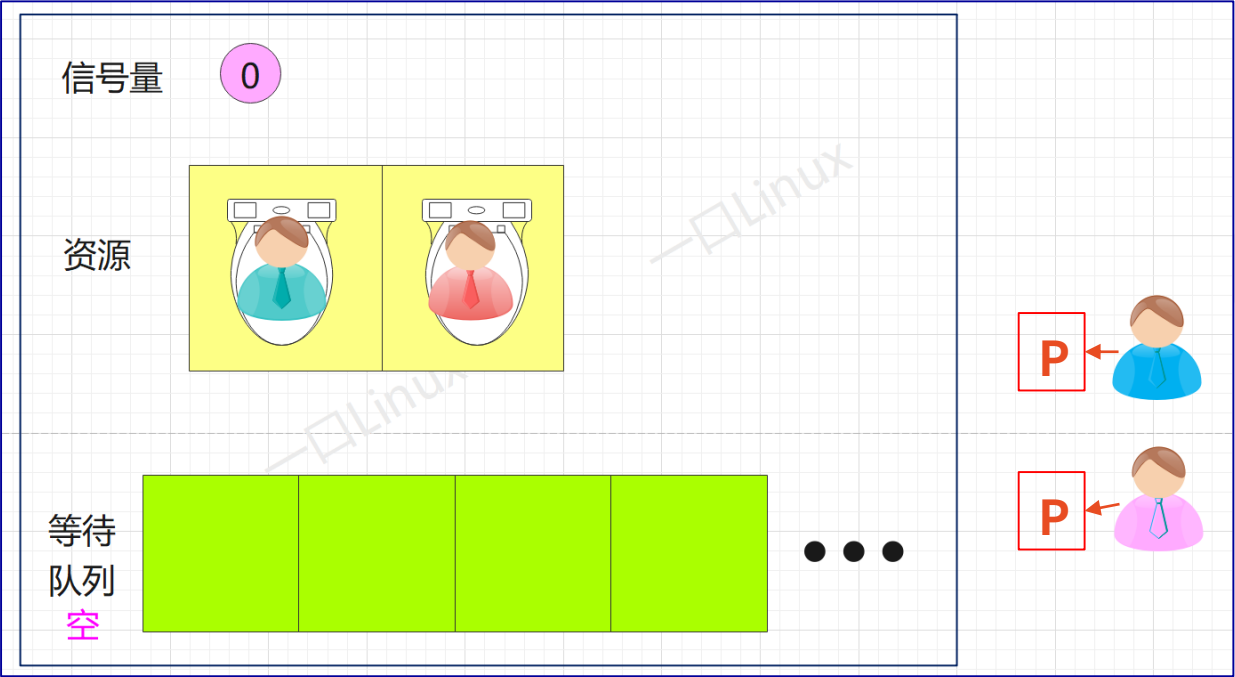
PV操作

- 通常把信号量操作抽象成PV操作
- P
 - 等待操作是等待信号灯的值变为大于0，然后将其减1；
- V
 - 释放操作则相反，用来唤醒等待资源的进程或者线程

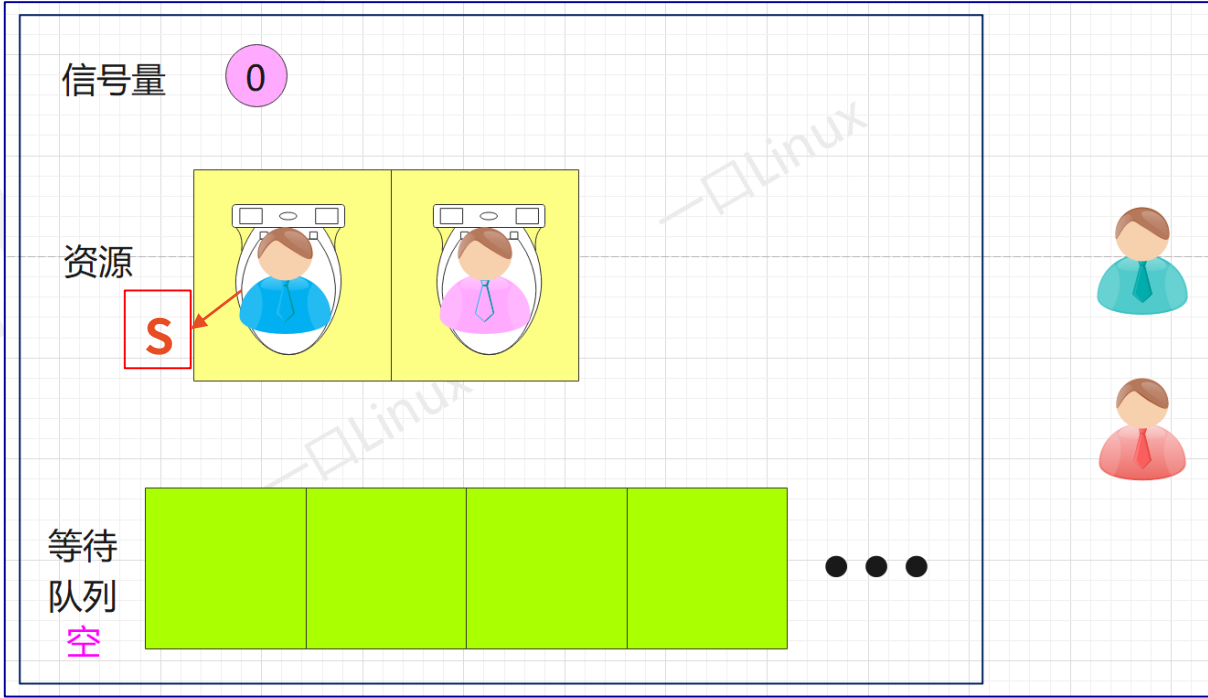
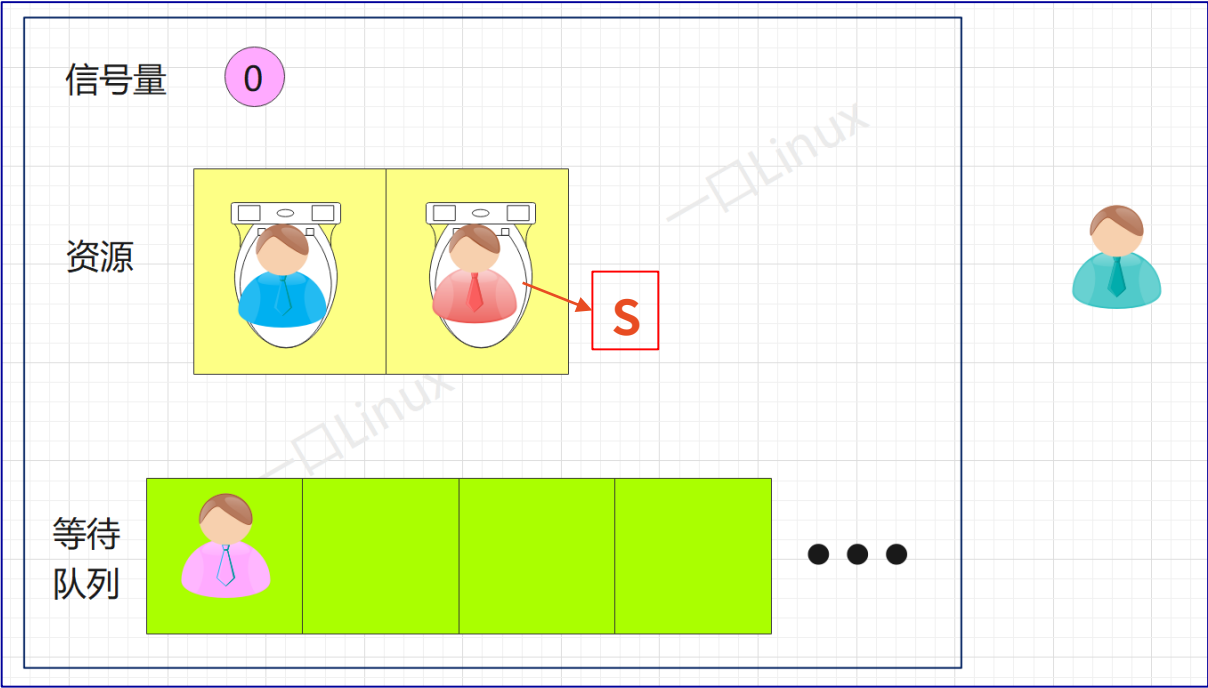
厕所模型1



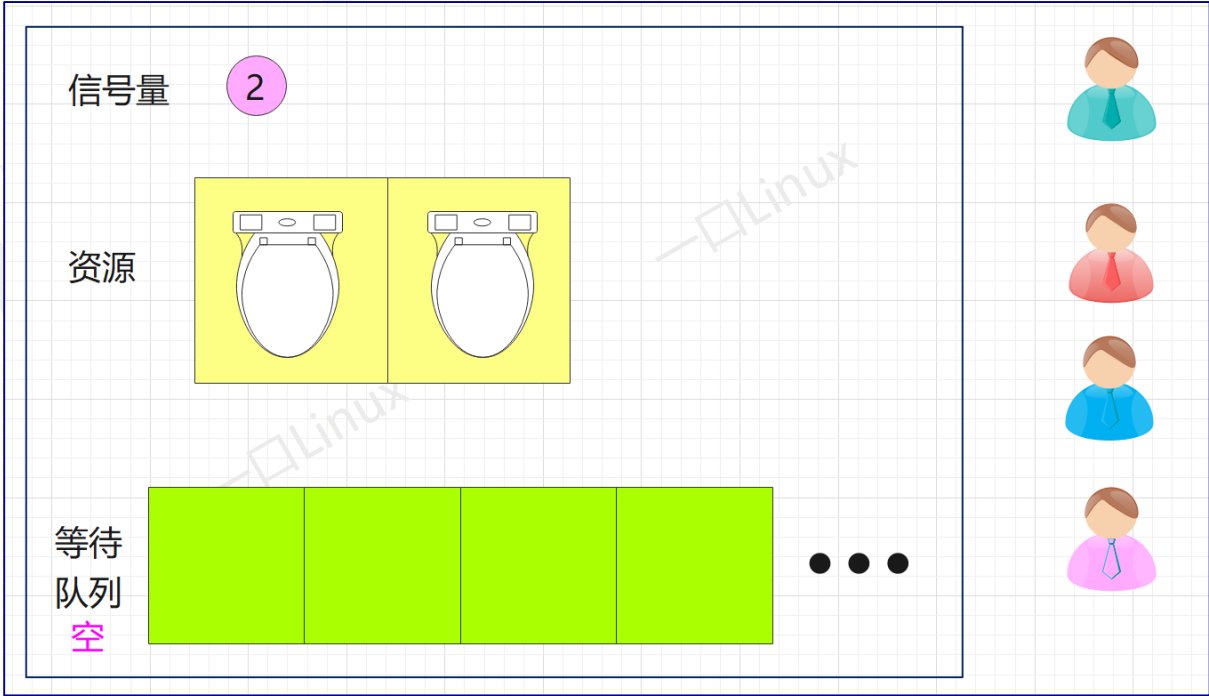
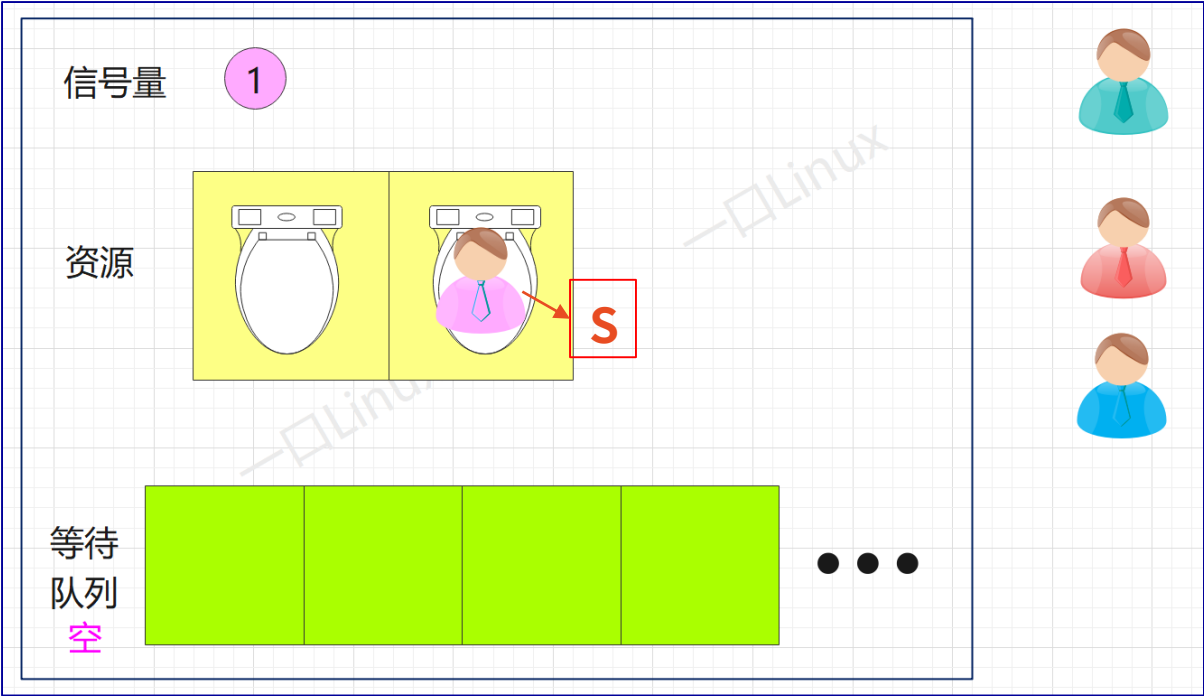
厕所模型2



厕所模型3



厕所模型4



sem函数-semget

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h></pre>
函数原型	<pre>int semget(key_t key, int nsems, int semflg);</pre>
函数参数	key: 和信号灯集关联的key值
	nsems: 信号灯集中包含的信号灯数目
	semflg: 信号灯集的访问权限, 通常为IPC_CREAT 0666
函数返回值	成功: 信号灯集ID
	出错: -1

sem函数-semctl

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h></pre>
函数原型	<pre>int semctl (int semid, int semnum, int cmd.../*union semun arg*/);</pre>
函数参数	<p>semid: 信号灯集ID semnum: 要修改的信号灯编号</p> <p>cmd: GETVAL: 获取信号灯的值 SETVAL: 设置信号灯的值 IPC_RMID: 从系统中删除信号灯集合</p> <pre>union semun { short val; /*SETVAL用的值*/ struct semid_ds* buf; /*IPC_STAT、IPC_SET用的semid_ds结构*/ unsigned short* array; /*SETALL、GETALL用的数组值*/ struct seminfo *buf; /*为控制IPC_INFO提供的缓存*/ } arg;</pre>
函数返回值	<p>成功: 0</p> <p>出错: -1错误原因存于errno中</p>

sem函数-semop

所需头文件	<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h></pre>
函数原型	<pre>int semop (int semid, struct sembuf *opsptr, size_t nops);</pre>
函数参数	<p>semid: 信号灯集ID</p> <pre>struct sembuf { short sem_num; // 要操作的信号灯的编号 short sem_op; // 0: 等待, 直到信号灯的值变成0 // 1: 释放资源, V操作 // -1: 分配资源, P操作 short sem_flg; // 0, IPC_NOWAIT, SEM_UNDO };</pre> <p>nops: 要操作的信号灯的个数</p>
函数返回值	<p>成功: 0</p> <p>出错: -1</p>

为SEM_UNDO时, 它将使操作系统跟踪当前进程对这个信号量的修改情况, 如果这个进程在没有释放该信号量的情况下终止, 操作系统将自动释放该进程持有的信号量。

函数使用

1. 初始化

```
int init_sem(int semid, int num, int val)
{
    union semun myun;
    myun.val = val;
    if(semctl(semid, num, SETVAL, myun) < 0)
    {
        perror("semctl");
        exit(1);
    }
    return 0;
}
```

2. 完成对信号量的P/V操作

```
int sem_p(int semid, int num)
{
    struct sembuf mybuf;
    mybuf.sem_num = num;
    mybuf.sem_op = -1;
    mybuf.sem_flg = SEM_UNDO;
    if(semop(semid, &mybuf, 1) < 0)
    {
        perror("semop");
        exit(1);
    }
    return 0;
}
```

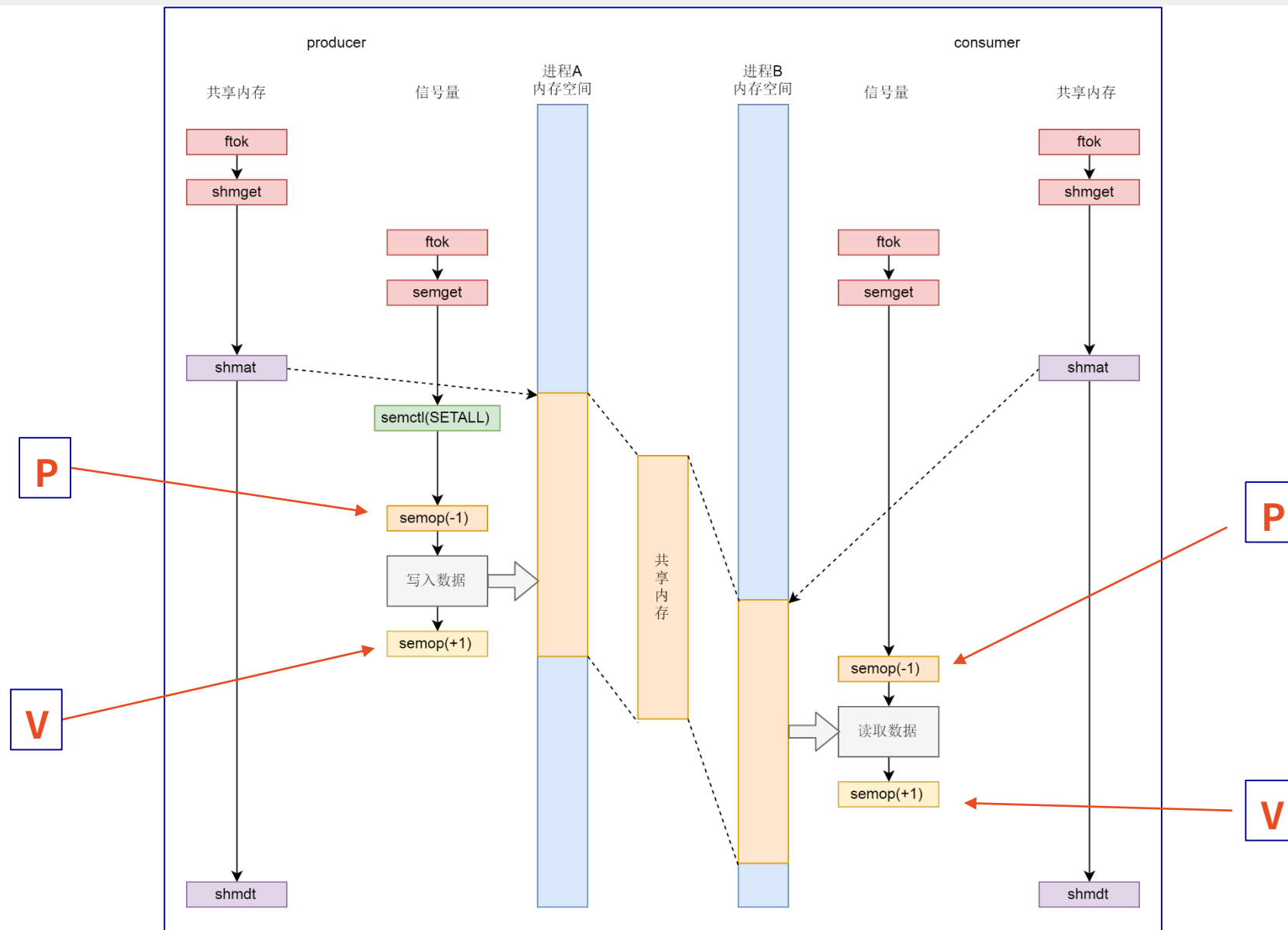
```
int sem_v(int semid, int num)
{
    struct sembuf mybuf;
    mybuf.sem_num = num;
    mybuf.sem_op = 1;
    mybuf.sem_flg = SEM_UNDO;
    if(semop(semid, &mybuf, 1) < 0)
    {
        perror("semop");
        exit(1);
    }
    return 0;
}
```

举例

- 物联网实训项目所有资料\3.课程代码讲解实例\3.linux系统编程\3.process\sem

一口Linux

使用案例:信号量与共享内存



08

信号

本项目中信号的使用场景

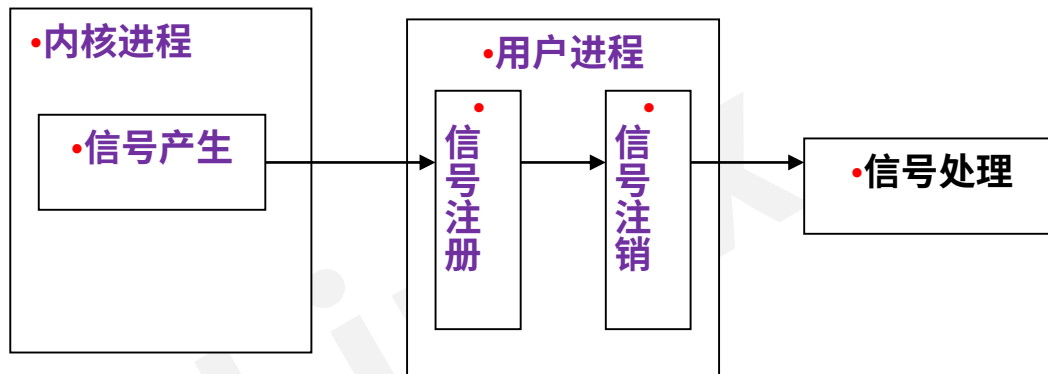
- 当我们按下ctrl+c终止主控程序时，希望能够实现如下操作：
 - 1.进程退出
 - 2.释放所有申请的资源
 - 互斥锁、条件变量、消息队列、共享内存、信号量、线程、设备文件描述符(摄像头、串口)

信号通信

- 信号是在软件层次上对**中断机制的一种模拟**，是一种异步通信方式
- 信号可以直接进行用户空间进程和内核进程之间的交互，内核进程也可以利用它来**通知**用户空间进程发生了哪些系统事件。
- 如果该进程当前并未处于执行态，则该信号就由内核保存起来，直到该进程恢复执行再传递给它；
- 如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消时才被传递给进程

信号通信

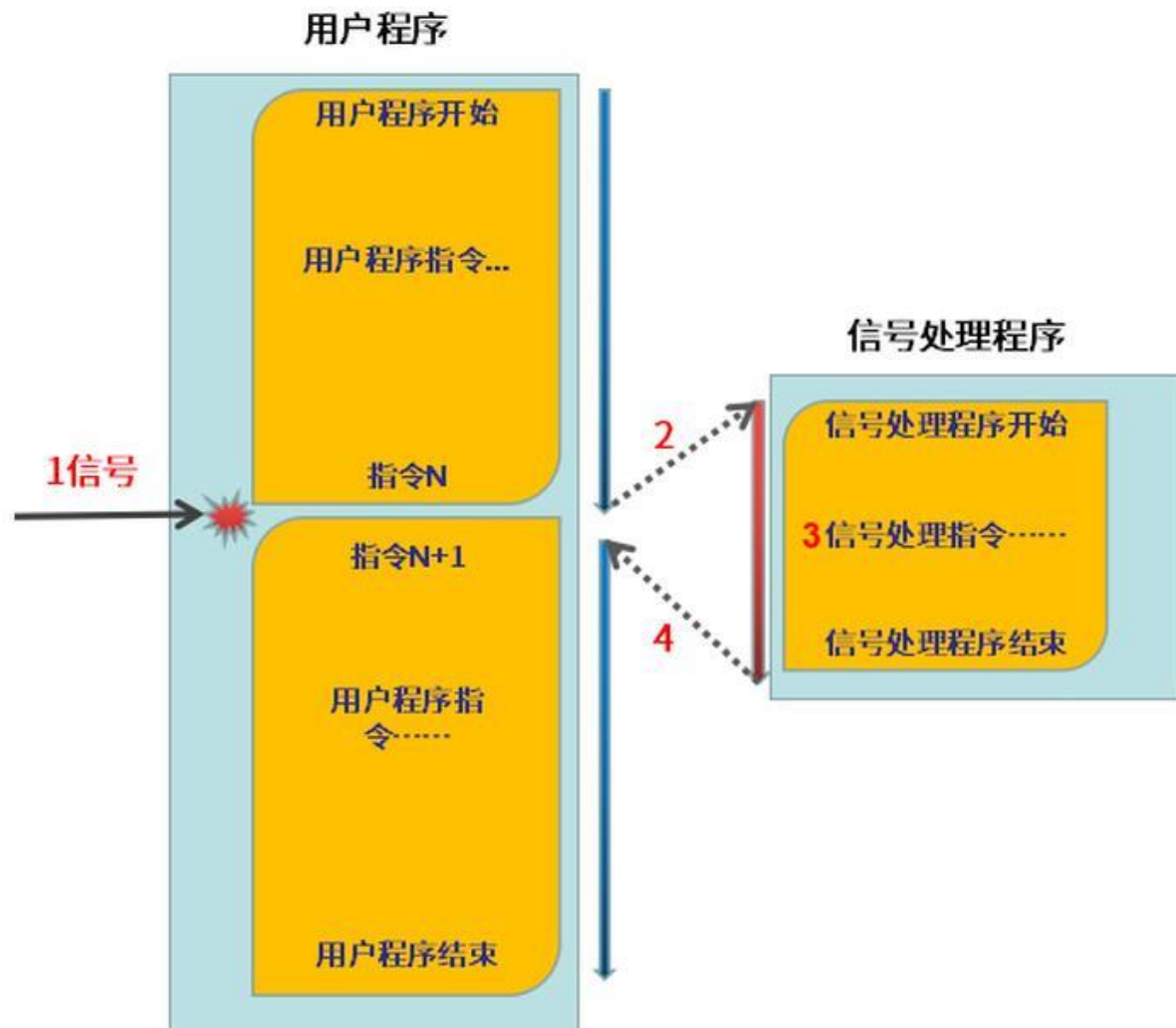
• 信号的生存周期



• 用户进程对信号的响应方式:

- 忽略信号:
 - 对信号不做任何处理，但是有两个信号不能忽略：即SIGKILL及SIGSTOP。
- 捕捉信号:
 - 定义信号处理函数，当信号发生时，执行相应的处理函数。
- 执行缺省操作:
 - Linux对每种信号都规定了默认操作

信号处理流程



信号类型

信号名	含义	默认操作
SIGHUP	该信号在用户终端连接(正常或非正常)结束时发出, 通常是在终端的控制进程结束时, 通知同一会话内的各个作业与控制终端不再关联。	终止
SIGINT	该信号在用户键入INTR字符(通常是Ctrl-C)时发出, 终端驱动程序发送此信号并送到前台进程中的每一个进程。	终止
SIGQUIT	该信号和SIGINT类似, 但由QUIT字符(通常是Ctrl-\\)来控制。	终止
SIGILL	该信号在一个进程企图执行一条非法指令时(可执行文件本身出现错误, 或者试图执行数据段、堆栈溢出时)发出。	终止
SIGFPE	该信号在发生致命的算术运算错误时发出。这里不仅包括浮点运算错误, 还包括溢出及除数为0等其它所有的算术的错误。	终止

信号类型

信号名	含义	默认操作
SIGKILL	该信号用来立即结束程序的运行，并且不能被阻塞、处理和忽略。	终止
SIGALRM	该信号当一个定时器到时的时候发出。	终止
SIGSTOP	该信号用于暂停一个进程，且不能被阻塞、处理或忽略。	暂停进程
SIGTSTP	该信号用于暂停交互进程，用户可键入SUSP字符（通常是Ctrl-Z）发出这个信号。	暂停进程
SIGCHLD	子进程改变状态时，父进程会收到这个信号	忽略
SIGABORT	该信号用于结束进程	终止

信号发送

- **kill()和raise()**

- **kill函数同读者熟知的kill系统命令一样，可以发送信号给进程或进程组（实际上，kill系统命令只是kill函数的一个用户接口）。**
- **kill -l 命令查看系统支持的信号列表**
- **Raise 函数允许进程向自己发送信号**

设置信号的处理方式

- 一个进程可以设定对信号的相应方式
- 信号处理的主要方法有两种
 - 使用简单的signal()函数
 - 使用信号集函数族sigaction
- signal()
 - 使用signal函数处理时，需指定要处理的信号和处理函数
 - 使用简单、易于理解

信号设置函数-signal

所需头文件	#include <signal.h>	
函数原型	void (*signal(int signum, void (*handler)(int)))(int);	
函数传入值	signum:	指定信号
	handler:	SIG_IGN: 忽略该信号。
		SIG_DFL: 采用系统默认方式处理信号。
		自定义的信号处理函数指针
函数返回值	成功: 设置之前的信号处理方式	
	出错: -1	

信号举例

- 物联网实训项目所有资料\3.课程代码讲解实例\3.linux系统编程\3.process\signal\signal.c

一口Linux

进程间通讯方式比较

- signal: 唯一的异步通信方式
- msg: 常用于cs模式中，按消息类型访问，可有优先级
- shm: 效率最高(直接访问内存)，需要同步、互斥机制
- sem: 配合共享内存使用，用以实现同步和互斥
- pipe: 具有亲缘关系的进程间，单工，数据在内存中
- fifo: 可用于任意进程间，双工，有文件名，数据在内存



更多嵌入式Linux知识
请关注一口君的公众号：一口Linux

公众号：一口Linux