调研报告：随机数发生器熵源调研

完成者姓名：杨一凡 学号：520021911080

1．随机数发生器：

1.1．伪随机数发生器与真随机数发生器：

伪随机数发生器为算法型发生器，生成的伪随机数有一定的随机特性，但这些数字序列并非传统意义上的随机数，相比于真随机数发生器，其一般运算更快，具有很好的统计特性，伪随机性序列满足 Golomb 随机性假设：

(1) 在每一个周期内，0 的个数与 1 的个数近似相等；

(2) 在每一个周期内，长度为 i 的游程数占游程总数的$\frac{1}{2^i}$；

(3) 定义自相关函数为

$$C(\tau)=\sum_{i=1}^{n}(-1)^{a_i+a_{i+\tau}}=\begin{cases}n, & \tau\equiv 0 \bmod n\\ c, & else\end{cases}$$

其中 c 为一个常数；

只有满足 Golomb 随机性假设，生成的序列才为伪随机序列。

真随机数发生器使用物理或者非物理的随机源，其并不可预测，一般具有次优的统计特性，但相比于真随机数发生器，其运算相对较慢。

1.2．Linux 中的伪随机数发生器：

1． rand()函数：

Linux 系统中可以使用 rand()函数进行伪随机数的生成，rand()函数的底层数学逻辑为线性同余，其生成的并不为真正的随机数，因为其周期很长，所以在一定的范围内可以看成随机的。rand()函数并不需要参数，其将返回 0 到 RAND_MAX 之间的任意整数。

2． srand()函数：

srand()函数为初始化随机数发生器，用于设置 rand()产生伪随机数时的种子。传入的参数 seed 为 unsigned int 类型，只要每次使用相同的 seed 值，就会得到相同的伪随机数列，常常可以使用 time()的返回值来改变 seed，从而可以得到不同的伪随机数序列，但 time()的返回值为可预测的，不存在不确定的熵源，因此只能生成伪随机数。

2．Linux 中的真随机数发生器：

2.1．真随机数发生器使用的熵源：

为生成统计意义上的随机数，需要一个不可预测的外部熵源，Linux 内核将使用计算机使用者做为熵源，使用者在使用计算机时敲击键盘的时间间隔，移动鼠标的距离与间隔，特定中断的时间间隔等等，对于计算机而言均为非确定和不可预测的。

计算机本身的行为完全由程序所控制，但操作者对于外设硬件的操作具有很大的不确定性，这些不确定性可以通过驱动程序中注册的中断处理例程（ISR）获取，内核根据这些非确定的设备事件

维护一个熵池，池中的数据为完全随机的，当有新的设备事件到来时，内核会估计新加入数据的随机性，当我们从熵池中取出数据时，内核会减少熵的估计值。

2.2. 真随机数发生器内核代码实现：

（1）熵池数据操作：

操作的不确定性可以通过驱动程序中注册的中断处理例程（ISR）获取，对应的函数为：

```
asmlinkage int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                                struct irqaction *action)
{
    int status = 1;
    int retval = 0;

    if (!(action->flags & SA_INTERRUPT))
        local_irq_enable();

    do
    {
        status |= action->flags;
        retval |= action->handler(irq, action->dev_id, regs);
        action = action->next;
    }while (action);

    if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);

    local_irq_disable();
    return retval;
}
```

若 ISR 在注册期间指定了 SA_SAMPLE_RANDOM 标志，在处理完 action 之后，还需要调用 add_interrupt_randomness()函数，其使用中断间隔时间为内核随机数发生器生成熵，进而可在熵池中填充新数据。

若计算机使用者完全不操作计算机，即作为熵源的产生者，完全不操作外部设备，不让熵池获得新的数据，则内核在每次从熵池中取数据后均会减少熵的估计值，若估计值为 0，则内核可以拒绝用户对于随机数的请求操作。

（2）获取内核随机数：

A．共存在两种方法可以从熵池中获取内核随机数，第一种为使用内核的随机数接口：

```
/*
 * This function is the exported kernel interface.  It returns some
 * number of good random numbers, suitable for key generation, seeding
 * TCP sequence numbers, etc.  It does not rely on the hardware random
 * number generator.  For random bytes direct from the hardware RNG
 * (when available), use get_random_bytes_arch().
 */
void get_random_bytes(void *buf, int nbytes)
{
#if DEBUG_RANDOM_BOOT > 0
    if (unlikely(nonblocking_pool.initialized == 0))
        printk(KERN_NOTICE "random: %pF get_random_bytes called "
               "with %d bits of entropy available\n",
               (void *) _RET_IP_,
               nonblocking_pool.entropy_total);
#endif
    trace_get_random_bytes(nbytes, _RET_IP_);
    extract_entropy(&nonblocking_pool, buf, nbytes, 0, 0);
}
```

get_random_bytes 为熵的输出接口，该函数返回长度为 nbytes 字节的缓冲区 buf，无论熵是否为 0 都将返回数据。

get_random_bytes 为一个系统调用，因此并不可在用户空间内直接调用。

B．第二种获取内核随机数的方法为/dev/random & /dev/urandom：

/dev/random 与/dev/urandom 均为 Linux 中的字符设备文件，其可作为伪随机数生成器，为系统提供随机数，Linux 系统中共存在三个熵池：主熵池、次熵池和 urandom 熵池，其中主熵池接受接收环境数据，大小为 512 字节，其为次熵池和 urandom 熵池提供随机数，两个文件设备分别对应着次

熵池和 urandom 熵池，其中次熵池大小为 128 字节，并且是阻塞的，urandom 熵池大小同样为 128 字节，但其为非阻塞的，用户可在用户空间进行字符设备文件/dev/random 与/dev/urandom 的访问。

当分别调用/dev/random 与/dev/urandom 请求 N 个随机二进制数，其处理流程如下：

/dev/random：

a. 次熵池计数器减去 N，若结果大于等于 0，直接从次熵池中取出 N 个随机二进制位并返回；

b. 若结果小于 0，先从次熵池中提取剩余所需的随机二进制位，主熵池计数器减去相应的值，同时返回 N 个随机二进制位；

c. 若此时主熵池和次熵池二者计数器之和都不够 N，则读取随机二进制位的动作会被阻塞，直到次熵池中有足够的随机二进制位；

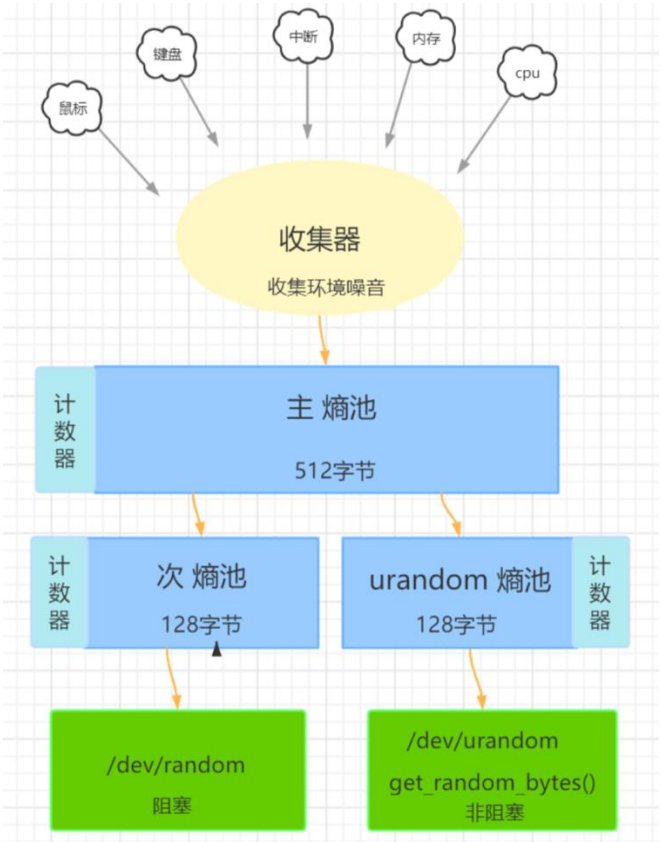
由/dev/random 的处理流程可知，/dev/random 返回指定数量的随机数，并且产生随机数的质量很高，属于真随机数，主要用于需要高质量随机数的场景。

/dev/urandom：

a . urandom 熵池计数器减去 N，若结果大于等于 0，直接从 urandom 熵池中取出 N 个随机二进制位并返回；

b . 若结果小于 0，请求不会阻塞，只不过返回 N 个伪随机二进制位，通过算法计算而来，其质量没有从 urandom 熵池中提取出的随机二进制位高；

由/dev/urandom 的处理流程可知，/dev/urandom 返回指定请求数量的随机数，如果请求的数量非常庞大的话，返回的随机数可能是伪随机数，随机数质量稍差些，即使如此，它们对大多数应用来说已经足够了。



参考文献

[1] https://blog.csdn.net/m0_74282605/article/details/128017996
[2] https://blog.csdn.net/hhhhhyyyyy8/article/details/102885457
[3] https://blog.csdn.net/fengye_csdn/article/details/120843570