1. 代码如下：

```python
import matplotlib as mpl
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator

mpl.rcParams['font.sans-serif'] = ['Times New Roman']  # 指定默认字体
mpl.rcParams['font.size'] = 14  # 字体大小
mpl.rcParams['axes.unicode_minus'] = False  # 正常显示负号

li = [0, 9, 8, 4, 4, 3, 6, 5, 1, 5, 0, 2, 1, 1, 1, 1, 8, 8, 5, 3, 9, 8, 9,
      9, 6, 1, 8, 4, 6, 4, 3, 7, 1, 3, 2, 9, 8, 6, 2, 9, 2, 7, 2, 7, 8, 4,
      2, 3, 0, 1, 9, 4, 7, 1, 5, 9, 1, 7, 3, 4, 3, 7, 1, 0, 3, 5, 9, 9, 4,
      9, 6, 1, 7, 5, 9, 4, 9, 7, 3, 6, 7, 7, 4, 5, 3, 5, 3, 1, 5, 6, 1, 1,
      9, 6, 6, 4, 0, 9, 4, 3]


def opt(frame_num):
    using = []
    fault = 0
    for i in range(len(li)):
        if li[i] in using:
            continue
        fault += 1
        if len(using) < frame_num:
            using.append(li[i])
        else:
            future_list = li[i + 1:]
            replace = using[0]
            next_use_time = 0
            for j in using:
                if j in future_list:
                    if future_list.index(j) > next_use_time:
                        replace = j
                        next_use_time = future_list.index(j)
                else:
                    replace = j
                    break
            using.remove(replace)
            using.append(li[i])
    return fault

```

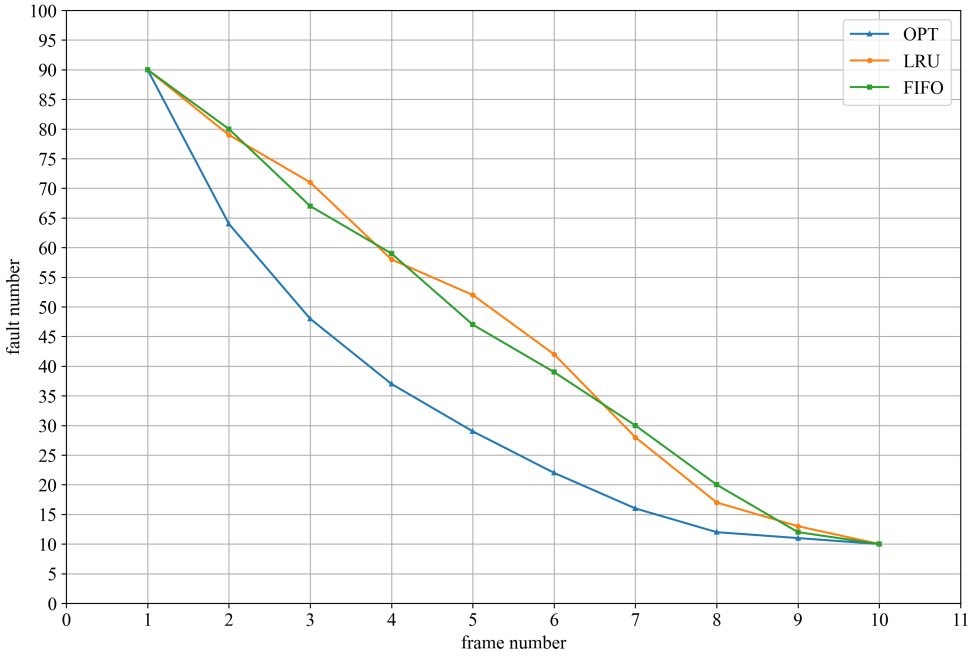
```
41
42  def lru(frame_num):
43      using = []
44      fault = 0
45      for i in range(len(li)):
46          if li[i] in using:
47              using.remove(li[i])
48              using.append(li[i])
49              continue
50          fault += 1
51          if len(using) < frame_num:
52              using.append(li[i])
53          else:
54              using.remove(using[0])
55              using.append(li[i])
56      return fault
57
58
59  def fifo(frame_num):
60      using = []
61      fault = 0
62      for i in range(len(li)):
63          if li[i] in using:
64              continue
65          fault += 1
66          if len(using) < frame_num:
67              using.append(li[i])
68          else:
69              using.remove(using[0])
70              using.append(li[i])
71      return fault
72
73
74  frame_num_list = [i for i in range(1, 11)]
75  opt_list = [opt(i) for i in range(1, 11)]
76  lru_list = [lru(i) for i in range(1, 11)]
77  fifo_list = [fifo(i) for i in range(1, 11)]
78  print('opt:', opt_list)
79  print('lru:', lru_list)
80  print('fifo:', fifo_list)
81
82  plt.figure(figsize=(12, 8), dpi=300)
83  a, = plt.plot(frame_num_list, opt_list, marker='^', ms=3)
84  b, = plt.plot(frame_num_list, lru_list, marker='o', ms=3)
```

```
c, = plt.plot(frame_num_list, fifo_list, marker='s', ms=3)
plt.legend(handles=[a, b, c], labels=[u'OPT', u'LRU', u'FIFO'])
plt.grid()

x_major_locator = MultipleLocator(1)
y_major_locator = MultipleLocator(5)
ax = plt.gca()
ax.xaxis.set_major_locator(x_major_locator)
ax.yaxis.set_major_locator(y_major_locator)

plt.xlim((0, 11))
plt.ylim((0, 100))
plt.xlabel(u'frame number')
plt.ylabel(u'fault number')
# plt.show()
plt.savefig('fault.png')
```

输出如下：

```
opt: [90, 64, 48, 37, 29, 22, 16, 12, 11, 10]
lru: [90, 79, 71, 58, 52, 42, 28, 17, 13, 10]
fifo: [90, 80, 67, 59, 47, 39, 30, 20, 12, 10]
```

图像如下：



2. 在32位4GB地址空间中采用二级页表机制，页内偏移占12位，代表着每一页大小为4KB，因此4GB地址空间一共被分为2^{20}个页，每个页需要一个页表项，而页表空间占用了4MB，因此每个页表项占用了4B。采用二级页表机制，页目录（第一级页表）占用10位地址，可以检索到1024个页表，每个页表检索1024个页面，因而第二级页表占用10位地址。页目录本身占用4KB空间，说明每个页目录项与每个页表项一样，占用4B内容，而页目录本身恰好是一个页面的大小，因此可以使用页目录自映射机制，将页目录恰好安置在一个页面上，该页称作页目录页；每个页表也恰好安置在一个页面上，该页称作页表页。

   将4MB的第二级页表看作对整个4GB内存空间的压缩映射，其起始地址是0x80000000，那么所有1024个第二级级页表页中，必然有一个页表页映射的是4MB第二级页表空间，这个页表页和页目录的功能和内容完全重合，将其作为页目录即可节省4KB一个页面的页目录空间，这就是页目录自映射机制。

   我们已经知道4MB第二级页表空间对于整个4GB内存空间是线性映射的，其起始地址是0x80000000，那么要寻找页目录起始地址，只需要知道页目录所映射的内存相对于整个内存布局在什么位置，再加上第二季页表起始地址即可。页目录所映射的就是第二级页表空间，其起始地址是0x80000000，其之前有0x80000000 >> 12 = 0x80000个页，每个页在页表中占4B的页表项空间，所以页目录所映射的内存相对于整个内存在第二级页表中的偏移量为0x80000 * 4 = 0x200000。页目录起始地址即为 0x80000000 + 0x200000 = 0x80200000。

3. 1. 32位地址，进程地址空间共4GB；页内偏移量12位，每一页有4KB大小。

   2. 1. 0x0 有效标志为0，页面尚未装入，引发缺页中断。
      2. 0x00803004 = 0b0000000010 0000000011 000000000100，页目录位0b0000000010 = 2。查页目录项2：有效标志为1，页表物理地址0x5000。原地址页表位0b0000000011 = 3，查0x5000处的页表项3：0x20001，有效标志为1，页面物理地址为0x20000。原地址页内偏移位0b000000000100 = 4，系统访问到的数据为0x0。
      3. 0x00402001 = 0b0000000001 0000000010 000000000001，页目录位0b0000000001 = 1，查页目录项1：有效标志为1，页表物理地址0x1000。原地址页表位0b0000000010 = 2，查0x1000处，即页目录本身的页表项（页目录项）2：0x5001，有效标志为1，页面物理地址为0x5000。原地址页内偏移位0b000000000001，系统访问到的数据为0x0。
   3. 访问物理地址0x326028，该物理地址的低12位0x028是页内偏移，物理页框号0x326000，可知虚拟地址页内偏移位为0b000000101000。在上图所示的页表内容中查到0x20000处页表的1偏移处存有该物理页框号且有效位为1，可知虚拟地址的页表偏移位为1。从页目录中查到页目录项3中页表物理地址为0x20000且标志位为1，可知虚拟地址页目录位为0b0000000011。综上，虚拟地址为0b0000000011 0000000001 000000101000，即0x00c01028。