OS第三次作业-刘兆薰 19373345

1. 在一个交换系统中，按内存地址排列的空闲区大小是: 10KB、4KB、20KB、18KB、7KB、 9KB、12KB和15KB。对于连续的段请求：12KB、10KB、9KB。使用FirstFit、BestFit、 WorstFit和NextFit将找出哪些空闲区?

FirstFit从前到后寻找第一个能够满足需求的内存块。12KB请求会找到3号20KB空闲区，分配后3号变为8KB；10KB请求会找到1号10KB空闲区，分配后1号空闲区全部占用；9KB请求会找到4号18KB空闲区，分配后4号变为9KB。FirstFit会使位置靠前的空闲区优先被分配，产生较多碎片。

BestFit寻找与请求最接近的空闲区进行分配。12KB请求会找到7号12KB空闲区，分配后7号空闲区全部被占用；10KB请求会找到1号10KB空闲区，分配后1号空闲区全部占用；9KB请求会找到6号9KB空闲区，分配后6号空闲区全部被占用。BestFit若能如上述情况找到恰好合适的空闲区就不产生碎片，否则容易产生较多碎片。

WorstFit每次都寻找最大的分区来分配。12KB请求会找到3号20KB空闲区，分配后3号变为8KB；10KB请求会找到4号18KB空闲区，分配后4号变为8KB；9KB请求会找到8号15KB空闲区，分配后8号变为6KB。WorstFit不容易产生很小的碎片，但却会使大块内存被分割，当大请求的作业到来时可能没有足够的大段连续空间。

NextFit每次从上一次分配空间的下一块空间开始查找，寻找第一个能够满足需求的空间。12KB请求会找到3号20KB空闲区，分配后3号变为8KB；10KB请求会找到4号18KB空闲区，分配后4号变为8KB；9KB请求会找到6号9KB空闲区，分配后6号空闲区全部被占用。NextFit不会像FirstFit那样让小碎片集中在内存前半部分，但也容易分割大块空闲区。

2. 解释页式（段式）存储管理中为什么要设置页（段）表和快表，简述页式（段式）地址转换过程。

页式存储管理的目的是高效利用内存，并实现大逻辑空间映射小物理空间。要使用页式内存管理，就需要存储一张页表，用于记录逻辑页向物理页的映射。由于查页表会增加一次访问内存，影响访问性能，人们设计了快表，即页表的cache，在查找页面映射时先查快表，若命中则无需查页表，若没有命中再访问页表，并做快表替换。

3. 叙述缺页中断的处理流程。

当进程执行过程中发生缺页中断时，需要进行页面换入，步骤如下：

1. 首先硬件会陷入内核，在堆栈中保存程序计数器。大多数机器将当前指令的各种状态信息保存在CPU中特殊的寄存器中；
2. 启动一个汇编代码例程保存通用寄存器及其它易失性信息，以免被操作系统破坏。这个例程将操作系统作为一个函数来调用；
3. 当操作系统发现是一个页面中断时，查找出来发生页面中断的虚拟页面（进程地址空间中的页面）。这个虚拟页面的信息通常会保存在一个硬件寄存器中，如果没有的话，操作系统必须检索程序计数器，取出这条指令，用软件分析该指令，通过分析找出发生页面中断的虚拟页面；
4. 检查虚拟地址的有效性及安全保护位。如果发生保护错误，则杀死该进程；

5. 操作系统查找一个空闲的页框(物理内存中的页面)，如果没有空闲页框则需要通过页面置换算法找到一个需要换出的页框；
6. 如果找的页框中的内容被修改了，则需要将修改的内容保存到磁盘上，此时会引起一个写磁盘调用，发生上下文切换（在等待磁盘写的过程中让其它进程运行）；
7. 页框干净后，操作系统根据虚拟地址对应磁盘上的位置，将保持在磁盘上的页面内容复制到“干净”的页框中，此时会引起一个读磁盘调用，发生上下文切换；
8. 当磁盘中的页面内容全部装入页框后，向操作系统发送一个中断。操作系统更新内存中的页表项，将虚拟页面映射的页框号更新为写入的页框，并将页框标记为正常状态；
9. 恢复缺页中断发生前的状态，将程序指令器重新指向引起缺页中断的指令；
10. 调度引起页面中断的进程，操作系统返回汇编代码例程；
11. 汇编代码例程恢复现场，将之前保存在通用寄存器中的信息恢复。

4. 假设一个机器有38位的虚拟地址和32位的物理地址：(1) 与一级页表相比，多级页表的主要优点是什么？(2) 如果使用二级页表，页面大小为16KB，每个页表项有4个字节。应该为虚拟地址中的第一级和第二级页表域各分配多少位?

1. 一级页表的缺陷在于，当逻辑地址空间很大的时候，页表本身会占用很多内存，查页表的过程也很耗费时间。因此人们设计多级页表机制，即为页表再设置页表，然后实行动态页表调入，即只将当前要用的页表项调入内存，其余的需要用时再调入。多级页表的优势在于节省了内存中存储页表的空间。
2. 页面大小为16KB，需要用14位来表示，即页内偏移占到地址的14位，对虚拟地址而言剩余24位。设剩余的24位中有n位用于查页表，(24-n)位用于查页目录。n位页表域可以表示2^n个虚拟页，每个虚拟页占用4个字节的页表项，则页表一共占用$2^n * 4B = 2^{n+2}B$空间，这些由页表占用的空间一共排在 $2^{n+2}B / 16KB = 2^{n-12}$ 个页面上，而这些页面需要由(n-12)位来表示，也就对应的是页目录的(24-n)位。此时有n-12 = 24-n，容易得n=18。因此虚拟地址中由6位表示页目录域，18位表示页表域，14位表示页内偏移。

5. 假设页面的访问存在一定的周期性循环，但周期之间会随机出现一些页面的访问。例 如：0,1,2...,511,431,0,1,2...511,332,0,1,2,...,511等。请思考：(1) LRU、FIFO和Clock算法的效果如何？ (2) 如果有500个页框，能否设计一个优于LRU、FIFO和Clock的算法?

1. 在题目所述得情况下，一共有512个可能得页面需要装载，因此若物理页框数大于512，那么不管哪种算法都不会产生缺页中断，LUR、FIFO和Clock算法得效果是相同的（事实上他们都没有应用过）。如果物理页框数不足512个，那么三种算法的效果也是几乎等同的。在首先装满所有物理页框之后，对于FIFO而言，在需要淘汰物理页的时候会从首先装入，即周期性循环中最靠前的物理页开始淘汰，不及该循环中的物理页再次出现，它一定会被淘汰，从而FIFO在应对这种情况的循环时，除了那个随机出现的页面之外，所有页面都会缺页。对于Clock算法而言情况也并不会太好，Clock在FIFO的原则之上给页面“第二次机会”，即如果一个页面装入后被再次访问过，就“再给它一次生存的机会”，但在上述情况中等待被淘汰的物理页大概率均不会有装入内存后的第二次访问机会，随机出现的页面可能会有再次访问从而享受到Clock的“庇护”，但大部分页面大概率都会缺页。对于LRU而言，它的效果也和前两者几乎等同，这是因为LRU淘汰最久没有用到的页面，在题目所述的循环中最久不用的页面和最先到来的页面几乎是等价的，因此LRU也会面临几乎所有页面都缺页的窘境。综上所述，面临题目所述的周期性页面访问时，LRU、FIFO和Clock的效果几乎一致，在页框足够时不会产生任何缺页，但在页框不足时几乎每次访问都会缺页，缺页率接近100%；

2. 针对周期性页面访问，在物理页框不足的情况下，使用与FIFO先进先出完全相对的“FILO 先进后出”（或后进先出）算法，其性能会大大优于LRU、FIFO和Clock。在最初的500个物理页被占满之后，到来的第500号虚拟页会替换掉499号，第501号又会替换500，等等，在此第500-511号虚拟页会全部缺页，但在新的循环到来时，前499个虚拟页会全部命中，夹在两次循环之间的随机页也会有500/512的概率命中，缺页率降至约为12/512=2.34%，远远优于LRU、FIFO和Clock。

6. 假设有10个页面，n个页框。页面的访问顺序为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。当n在[1,10]中取值时，请编写程序实现OPT、LRU、FIFO页面置换算法，并根据页面访问顺序模拟执行，分别计算缺页数量，画出缺页数量随页框数n的变化曲线（3条线）。

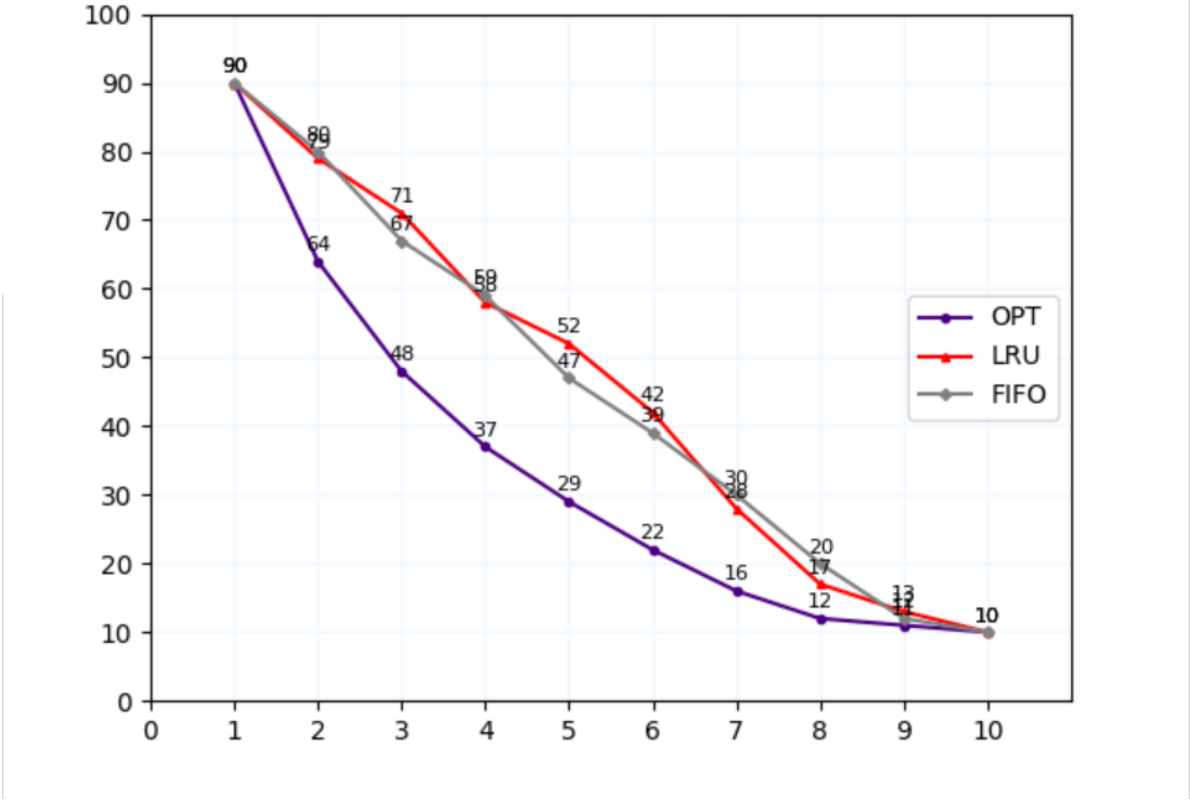
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def show_axis():
    plt.xlim(0, 11)
    plt.xticks([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    plt.ylim(0, 100)
    plt.yticks([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
    for i in range(10):
        plt.plot([0, 11], [(i+1)*10, (i+1)*10], linewidth='1',
color='aliceblue')
        plt.plot([i+1, i+1], [0, 100], linewidth='1', color='aliceblue')


def show_graph(opt_faults, lru_faults, fifo_faults):
    show_axis()
    ln1, = plt.plot(x, opt_faults, color='cornflowerblue', marker='o',
markersize='3')
    ln2, = plt.plot(x, lru_faults, color='gold', marker='^', markersize='3')
    ln3, = plt.plot(x, fifo_faults, color='firebrick', marker='D',
markersize='3')
    plt.legend([ln1, ln2, ln3], ['OPT', 'LRU', 'FIFO'])
    for i in range(10):
        plt.text(i + 1, opt_faults[i] + 1, str(opt_faults[i]), ha='center',
va='bottom', fontsize='8')
        plt.text(i + 1, lru_faults[i] + 1, str(lru_faults[i]), ha='center',
va='bottom', fontsize='8')
        plt.text(i + 1, fifo_faults[i] + 1, str(fifo_faults[i]), ha='center',
va='bottom', fontsize='8')
    plt.show()

page_visit_list = [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_to_replace(i, frame_usage):
    future_list = page_visit_list[i+1:]
    to_replace = frame_usage[0]
    next_use = 0
    for page in frame_usage:
        if page in future_list:
            if future_list.index(page) > next_use:
                to_replace = page
                next_use = future_list.index(page)
        else:
            to_replace = page
            break
    return to_replace


def opt(frame_num):
    page_fault = 0
    frame_usage = []
    to_replace = 0
    for i in range(len(page_visit_list)):
        page = page_visit_list[i]
        if page in frame_usage:
            to_replace = opt_to_replace(i, frame_usage)
            continue
        page_fault = page_fault + 1
        if len(frame_usage) < frame_num:
            frame_usage.append(page)
            to_replace = opt_to_replace(i, frame_usage)
            continue
        frame_usage[frame_usage.index(to_replace)] = page
        to_replace = opt_to_replace(i, frame_usage)
    return page_fault


def lru(frame_num):
    page_fault = 0
    frame_usage = []
    for page in page_visit_list:
        if page in frame_usage:
            frame_usage.remove(page)
            frame_usage.append(page)
            continue
        page_fault = page_fault + 1
        if len(frame_usage) < frame_num:
            frame_usage.append(page)
            continue
        frame_usage.remove(frame_usage[0])
        frame_usage.append(page)
    return page_fault


def fifo(frame_num):
```

```
    page_fault = 0
    frame_usage = []
    for page in page_visit_list:
        if page in frame_usage:
            continue
        page_fault = page_fault + 1
        if len(frame_usage) < frame_num:
            frame_usage.append(page)
            continue
        frame_usage.remove(frame_usage[0])
        frame_usage.append(page)
    return page_fault


if __name__ == '__main__':
    opt_faults, lru_faults, fifo_faults = [], [], []
    for i in range(10):
        opt_faults.append(opt(i+1))
        lru_faults.append(lru(i+1))
        fifo_faults.append(fifo(i+1))
    show_graph(opt_faults, lru_faults, fifo_faults)
    print(opt_faults)
    print(lru_faults)
    print(fifo_faults)
    print('Done.')
```

物理页框数	1	2	3	4	5	6	7	8	9	10
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

利用`Matplotlib`：



7. 一个32位的虚拟存储系统有两级页表，其逻辑地址中，第22到31位是第一级页表，12 位到21位是第二级页表，页内偏移占0到11位。一个进程的地址空间为4GB，如果从 0x80000000开始映射4MB大小页表空间，请问第一级页表所占4KB空间的起始地址？并说明理由。（注意B代表字节，一个32位地址占4字节）

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。

8. 一个32位的虚拟存储系统有两级页表，其逻辑地址中，第22到31位是第一级页表（页目录）的索引，第12位到21位是第二级页表的索引，页内偏移占第0到11位。每个页表（目录）项包含20位物理页框号和12位标志位，其中最后1位为页有效位。

虚拟地址格式：

10位	10位	12位
页目录号	二级页表号	页内偏移量

页目录项、页表项格式：

20位	11位	11位
物理页框号	其他页面标志	页面有效标志

请问进程整个的地址空间有多少字节？一页有多少字节？

如果当前进程的页目录物理基地址、页目录和相应页表内容如图下所示：

页目录物理地址：0x1000	页表物理地址：0x5000	页表物理地址：0x20000
0000: 0x0	0000: 0x0	0000: 0x9000
0001: 0x1001	0001: 0x4e001	0001: 0x326001
0002: 0x5001	0002: 0x67001	0002: 0x41001
0003: 0x20001	0003: 0x20001	0003: 0x0
0004: 0x0	0004: 0x0	0004: 0x0
…	…	…
1023: 0x0	1023: 0x0	1023: 0x0

描述访问以下虚拟地址时系统进行地址转换的过程，如可行给出最终访存获取到的数据。虚拟地址：0x0、0x00803004、0x00402001。条件如上，若要访问物理地址0x326028，需要使用哪个虚拟地址？

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。

根据上图的页目录和页表内容，访问0x0时先查页目录项0000：有效标志为0，页面尚未装入，引发缺页中断。

根据上图的页目录和页表内容，访问0x00803004 = 0b00000000100000000011000000000100，页目录位0b0000000010，查页目录项0002：有效标志为1，页表物理地址0x5000。原地址页表位0b0000000011，查0x5000处的页表项0003：0x20001，有效标志为1，页面物理地址为0x20000。原地址页内偏移位0b000000000100，系统为大端和小端访问到的数据分别为0b00000000和0b00000001。

根据上图的页目录和页表内容，访问0x00402001 = 0b00000000010000000010000000000001，页目录位0b0000000001，查页目录项0001：有效标志为1，页表物理地址0x1000。原地址页表位0b0000000010，查0x1000处，即页目录本身的页表项（页目录项）0002：0x5001，有效标志为1，页面物理地址为0x5000。原地址页内偏移位0b000000000001，系统为大端和小端访问到的数据都会是0b00000000。

若要访问物理地址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0001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0b0000000001。从页目录中查到页目录项0003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0b00000000110000000001000000101000，即0x00c01028。