操作系统第二次作业-请求调页存储管理方式模拟

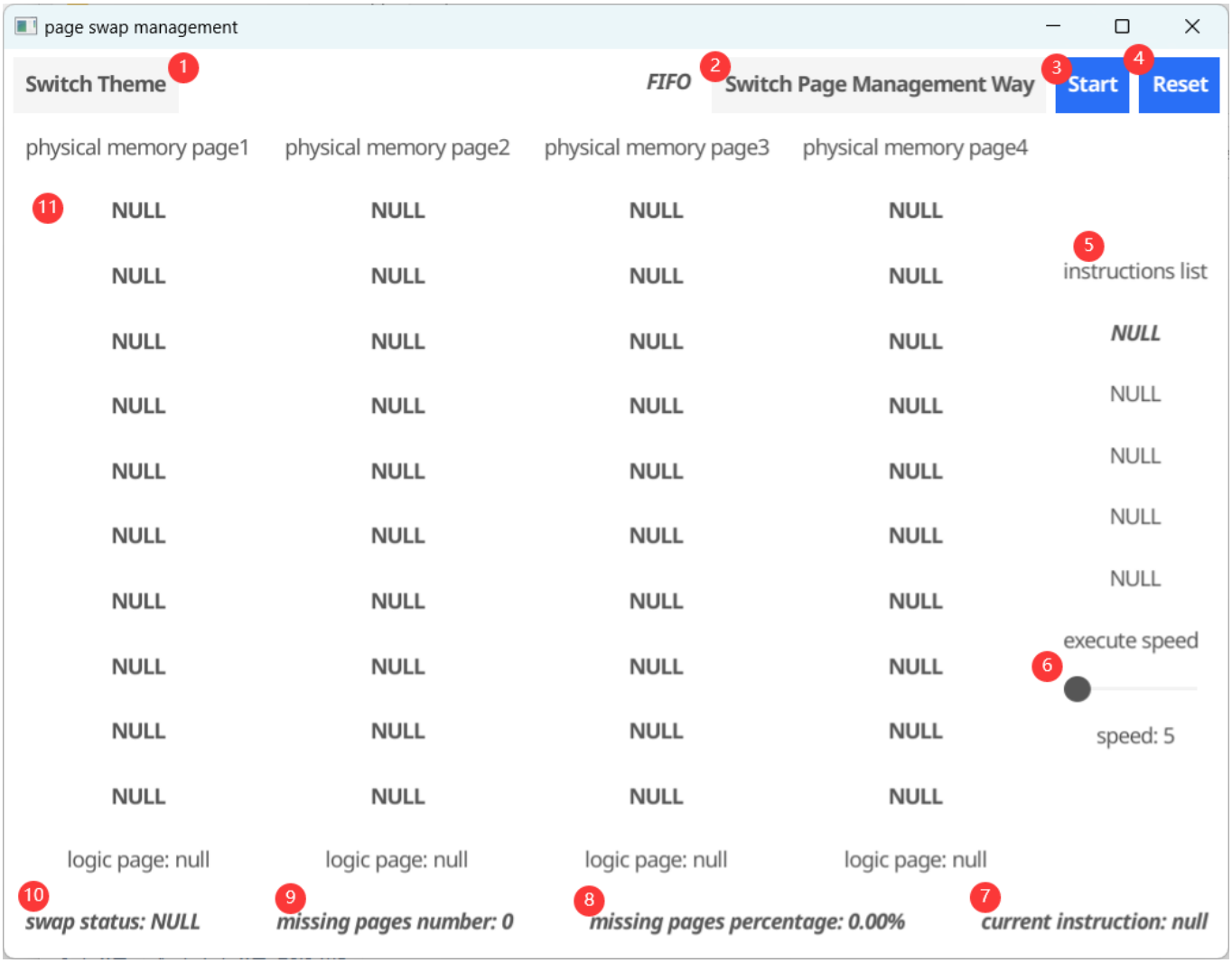
1. 介绍

本次项目采用`Go`语言编写，使用`fyne`GUI库，实现了对操作系统中请求调页存储管理方式的模拟，实现了`FIFO`和`LRU`两种调页的置换方式。

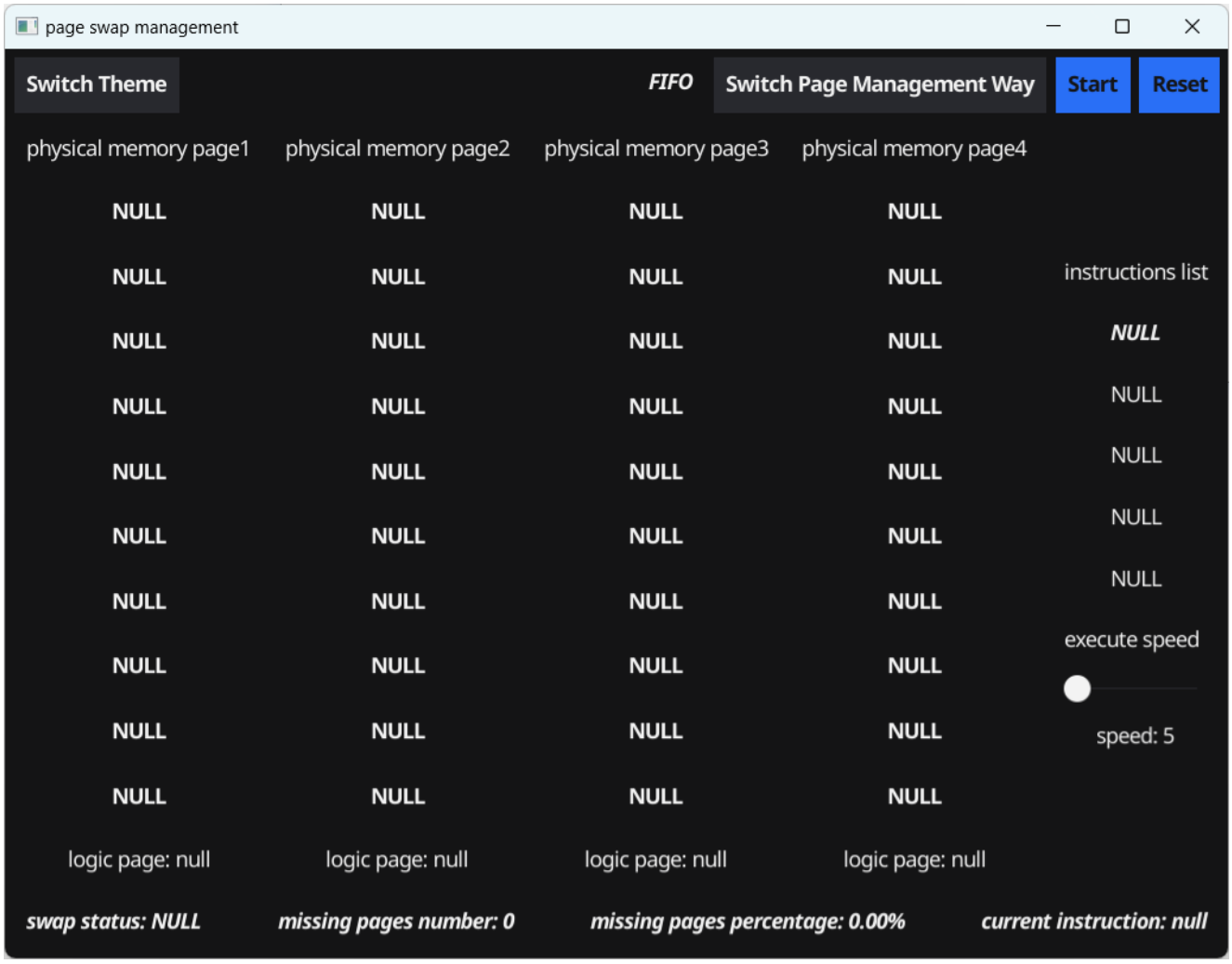
- 模拟方式：

  本项目模拟一个作业的执行过程，操作系统为该作业分配了4个内存块，该作业总共有320条指令，即它的地址空间为32页，目前所有页还没有调入内存。指令执行顺序为，50%的指令是顺序执行的，25%是均匀分布在前地址部分，25%是均匀分布在后地址部分。在模拟过程中，如果所访问指令在内存中，则显示其物理地址，并转到下一条指令；如果没有在内存中，则发生缺页，此时需要记录缺页次数，并将其调入内存。如果4个内存块中已装入作业，则需进行页面置换。所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。置换方式采用FIFO和LRU方式执行。

2. 界面设计



1. 切换主题按钮，可以切换白色与黑色主题，切换效果如下：



2. 切换置换页算法方式：可选FIFO与LRU算法
3. 开始按钮：点击后开始模拟过程
4. 重置按钮：一次模拟完成后，进行重置回复到初始状态，可以进行下一次模拟
5. 执行指令序列：展示当前即将执行的五个指令
6. 速度调节：调节模拟过程的速度
7. 当前指令：指示当前执行指令
8. 缺页百分比：指示缺页的百分比比率
9. 缺页数：指示缺页的数量
10. 交换状态：指令模拟一条指令执行过程的置换页面的状态
11. 展示区：展示物理内存块中指令存储情况

3. 代码设计

- FIFO:

```
func FIFO(instruction int) {
    if checkInMemory(instruction) {
        SwapStatusShowStr.Set("instruction " + strconv.Itoa(instruction) + " is in memory")
        return
    } else {
        MissingPagesNum++
        MissingPagesNumStr.Set("missing pages number: " + strconv.Itoa(MissingPagesNum))
        MissingPagesPercentageStr.Set("missing pages percentage: " + fmt.Sprintf("%.2f",
float64(MissingPagesNum)/float64(InstructionsNum)*100) + "%")
        lenth := MemoryQueue.Size()
        if lenth < MemorySize {
            MemoryQueue.Enqueue(lenth + 1)
            replacePage(instruction, lenth+1)
        } else {
            pos, _ := MemoryQueue.Dequeue()
            replacePage(instruction, pos.(int))
            MemoryQueue.Enqueue(pos)
        }
    }
}
```

使用一个 `MemoryQueue` 来存储使用物理内存的顺序，调入内存中时，入队被调入内存的位置，替换时选择替换出队列的位置，即可实现 `FIFO` 方法。

- LRU:

```go
func LRU(instruction int) {
    if checkInMemory(instruction) {
        SwapStatusShowStr.Set("instruction " + strconv.Itoa(instruction) + " is in memory")
        return
    } else {
        MissingPagesNum++
        MissingPagesNumStr.Set("missing pages number: " + strconv.Itoa(MissingPagesNum))
        MissingPagesPercentageStr.Set("missing pages percentage: " + fmt.Sprintf("%.2f",
float64(MissingPagesNum)/float64(InstructionsNum)*100) + "%")
        if !IsFull {
            Memory++
            MemoryFrequency[Memory-1] = LRUTime
            if Memory == MemorySize {
                IsFull = true
            }
            replacePage(instruction, Memory)
        } else {
            pos := 1
            for i := 0; i < MemorySize; i++ {
                if MemoryFrequency[i] < MemoryFrequency[pos-1] {
                    pos = i + 1
                }
            }
            MemoryFrequency[pos-1] = LRUTime
            replacePage(instruction, pos)
        }
    }
    LRUTime++ //时间戳加一
}
```

定义一个 `LRUTime` 用来记录程序执行的时间，`MemoryFrequence[4]` 记录每个内存位置最近被使用的时间，每进行一次指令执行，便将 `LRUTime++`，每当有内存块被使用时，便将对应的 `MemoryFrequence[pos]` 置为当前的 `LRUTime`，每次要发生替换时，比较每个内存块对应 `MemoryFrequence[pos]` 的大小，最小的即为最近未被使用的位置，挑选其进行置换，即实现了 `LRU` 替换。

- checkInMemory:

```go
func checkInMemory(instruction int) bool {
    for i := 0; i < MemorySize; i++ {
        for j := 0; j < PageSize; j++ {
            if MemoryButtons[i][j].Text == strconv.Itoa(instruction) {
                hilightButton(MemoryButtons[i][j])
                //LRU
                if PageManagementWay == 1 {
                    MemoryFrequency[i] = LRUTime
                }
                return true
            }
        }
    }
    return false
}
```

检查当前执行指令是否已经在内存中，若在，则高亮其位置，且若当前方法为 `LRU` 方法，则同时将对应 `MemoryFrequence[i]` 置为当前 `LRUTime` 。

- replacePage:

```go
// physicalPage: 1,2,3,4
func replacePage(instruction, physicalPage int) {
    logicPage := getPageByInstruction(instruction)
    SwapStatusShowStr.Set(fmt.Sprintf("swap logical page %d and physical page %d",
logicPage, physicalPage))
    LogicPageShowStr[physicalPage-1].Set("logic page: " + strconv.Itoa(logicPage))
    // if logicPage == 33 {
    //  fmt.Println("Instruction:", instruction)
    // }
    for i := 0; i < PageSize; i++ {
        if PageTable[logicPage-1][i] == instruction {
            hilightButton(MemoryButtons[physicalPage-1][i])
        }
        MemoryButtons[physicalPage-1][i].SetText(strconv.Itoa(PageTable[logicPage-1][i]))
    }
}
```

传入参数为当前执行指令与要替换的物理内存位置，首先计算得到当前指令所在的逻辑页号，再根据此逻辑页号取得该页中所有指令，并将其替换到物理内存中，同时高亮替换进来的指令。