22-23 Final

Q1

1. Can the **Priority-based scheduling** schemes result in starvation? If so. how might you fix this? Explain. (4 marks)

基于优先级的调度方案可能会导致饥饿现象。饥饿现象是指某些低优先级的进程由于高优先级进程持续占用资源而长期得不到调度，导致这些低优先级进程无法得到执行的机会。

为了解决饥饿问题，可以采取以下几种方法：

1. 老化机制：逐渐提高等待时间较长的低优先级进程的优先级，使其最终能获得调度的机会。通过这种方式，可以确保所有进程在合理的时间内得到执行。
2. 时间片轮转（Round Robin）调度：结合优先级和时间片轮转调度方式。在保证高优先级进程优先执行的同时，设置每个进程的最大连续执行时间。当高优先级进程超过时间片时，调度器切换到下一个进程，无论其优先级高低。这样可以避免某个进程长期占用CPU。
3. 多级队列调度：将进程分成多个优先级队列，不同队列有不同的时间片。高优先级进程优先调度，但每个队列都有自己的时间片，低优先级队列也能定期获得CPU时间。这种方式也能缓解饥饿问题。

通过这些方法，可以有效地避免基于优先级调度中的饥饿现象，确保系统资源的公平分配。

2. Give **two** positive and **two** negative effects of increasing the **page size**. Explain. (12 marks)

正面影响：

1. 减少页表开销：
   增大页面大小可以减少页表的项数，从而减小页表的开销。因为每个页表项对应一个固定大小的页面，页面越大，所需的页表项就越少，进而减少内存管理单元（MMU）的负担，提高内存管理的效率。

2. 提高磁盘I/O效率：
   页面大小增大后，每次页面换入换出的数据量增加，减少了磁盘I/O操作的频率。由于磁盘I/O操作较慢，减少I/O次数可以显著提高系统性能。

负面影响：

1. 增加内存碎片：
   增大页面大小会导致内存碎片问题更加严重。当分配一个大页面时，往往会出现未被利用的小块内存，造成内存浪费。尤其在处理大量小文件或小数据块时，大页面容易导致更多的内存碎片。
2. 增加页面换入换出开销：
   页面大小增大后，每次页面换入换出的数据量增加，虽然I/O操作次数减少，但每次I/O操作的时间会增加。特别是在页面换入换出频繁的情况下，大页面可能会导致换入换出的开销过大，影响系统性能。

3. How do **base register** and **limit register** help in protecting against illegal memory access? (4 marks)

保护机制：

1. 地址转换与验证：
   - 当进程试图访问内存时，CPU会将逻辑地址与基址寄存器的值相加，得到实际的物理内存地址。
   - 计算后的地址会与限长寄存器的值进行比较。如果地址偏移超出限长寄存器的值，则该访问请求被认为是非法的。
2. 防止越界访问：
   - 基址寄存器确保进程只能访问从基址起始的内存范围。
   - 限长寄存器确保进程不能访问超出其分配范围的内存。这防止了一个进程访问其他进程的内存或系统内存，从而保护系统的稳定性和安全性。

4. If you have to design and program a **web browser** (which is a complex application), what would you choose to use multiple processes, multiple threads, or both? Explain your answer. (6 marks)
Discuss **two** advantages of your choice. (6 marks)

如果要设计和编写一个复杂的应用程序，例如一个网页浏览器，我会选择使用多进程和多线程的组合方案。这种方法能充分利用两者的优势，提供更好的性能和稳定性。

解释选择：

- 多进程：用于隔离不同的功能模块，例如不同的网页标签、插件和渲染引擎。每个标签页或插件运行在独立的进程中，即使某个标签页崩溃，也不会影响整个浏览器的运行。
- 多线程：用于提高单个进程内的并发能力。例如，在渲染一个复杂的网页时，可以使用多个线程并行处理不同的任务，如页面布局、样式计算和JavaScript执行。

两个优势：

1. 稳定性和隔离性：
    - 使用多进程架构可以将不同的网页标签和插件隔离开来。如果一个标签页发生崩溃，其他标签页和浏览器主进程不会受到影响，从而提高了浏览器的整体稳定性。这种隔离还增强了安全性，因为不同进程之间的内存是隔离的，可以防止恶意网页或插件的攻击。
2. 并行处理和性能优化：
    - 多线程可以在单个进程内进行并行处理，提高处理效率。例如，在渲染复杂的网页时，可以将不同的渲染任务分配给多个线程并行执行，加快页面加载速度和响应速度。这种并行处理能力使得浏览器在处理复杂任务时更加流畅和高效。

5. **Redundant Array of Independent Disks** or **RAID** is a set of physical disk drives viewed by the operating system as a single logical drive. Give **two** advantages and **two** disadvantages of RAID. Explain. (12 marks)

RAID的两个优点：

1. 数据冗余和可靠性提高：
   RAID通过数据冗余提高了系统的可靠性。例如，RAID 1（镜像）和RAID 5（奇偶校验）等级别通过将数据复制到多个磁盘或计算奇偶校验信息，即使某个磁盘发生故障，数据也不会丢失。这样，可以保证系统在单个磁盘故障时仍然能够正常工作，从而提高了数据的安全性和系统的可靠性。
2. 性能提升：
   RAID可以通过并行读写多个磁盘来提升性能。例如，RAID 0（条带化）将数据分割成多个块，并行地写入到多个磁盘中，从而大大提高了读写速度。特别是在需要高性能的环境中，如数据库或文件服务器，RAID能够显著提高系统的I/O性能。

RAID的两个缺点：

1. 成本增加：
   RAID实现数据冗余和性能提升通常需要额外的硬件支持，例如多个硬盘、RAID控制器等。这些额外的硬件会增加系统的成本。此外，一些RAID级别（如RAID 1和RAID 10）需要将数据完全复制到另一个磁盘，导致实际可用存储空间减少，从而增加了存储成本。
2. 复杂性增加：
   RAID配置和维护相对复杂，特别是在高端RAID级别（如RAID 5和RAID 6）中，需要进行数据分割、奇偶校验计算和恢复等操作。对于系统管理员来说，管理RAID系统需要更多的知识和经验，同时在RAID系统发生故障时，数据恢复过程可能较为复杂和耗时。

Q2

1.

1. Consider a *Real-Time System* in which there are three processes. Their arrival time, period and execution time are as follows:

Process	Arrival time	Execution time	Deadline
P1	0	10	33
P2	4	3	28
P3	5	10	29

Calculate total utilization of CPU. (2 marks)

We assume that all three processes are released at time 0. Explain the **Earliest Deadline First Scheduling Algorithm** of the processes. (4 marks)

Show the processes on timing diagram. (2 marks)

总CPU利用率计算：

总CPU利用率 U 可以通过以下公式计算：

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

其中：

- C_i 是进程 P_i 的执行时间
- T_i 是进程 P_i 的周期（在此问题中假设每个进程的周期等于其截止时间）

对于给定的进程，我们有：

- $P1: C_1 = 10, T_1 = 33$
- $P2: C_2 = 3, T_2 = 28$
- $P3: C_3 = 10, T_3 = 29$

因此，总CPU利用率 U 计算如下：

$$U = \frac{10}{33} + \frac{3}{28} + \frac{10}{29}$$

我们来计算这个值：

$$U \approx \frac{10}{33} + \frac{3}{28} + \frac{10}{29} \approx 0.303 + 0.107 + 0.345 \approx 0.755$$

所以，总CPU利用率约为75.5%。

Earliest Deadline First (EDF) 调度算法解释：

Earliest Deadline First (EDF) 调度算法是一种动态优先级调度算法，用于实时系统。其主要思想是：

1. 动态优先级：每个任务的优先级是根据其最早截止时间确定的。截止时间越早，优先级越高。
2. 任务选择：调度器总是选择具有最早截止时间的任务进行调度执行。

在EDF算法中，每当有新任务到达或现有任务完成时，调度器会重新评估所有任务的截止时间，并选择优先级最高（即截止时间最早）的任务执行。这种方法可以保证在CPU利用率小于或等于100%的情况下，所有任务都能在截止时间之前完成。

时间图示例：

时间： 0 4 7 17 23

|------|------|------|------|

任务： P1 P2 P3 P1

2.

2. Calculate the number of page faults for the following sequence of page references (each element in the sequence represents a page number) using the **First-In, First-Out (FIFO) algorithm** with frame size of **3**.

1 2 3 2 1 5 2 1 6 2 5 6 3 1 3 6 1 2 4 3

(6 marks)

FIFO 页面替换过程：

1. 初始状态：
   - 帧: [空, 空, 空]
2. 处理页面引用1：
   - 页面1不在帧中，发生页面错误。
   - 帧: [1, 空, 空]
   - 页面错误数: 1
3. 处理页面引用2：
   - 页面2不在帧中，发生页面错误。
   - 帧: [1, 2, 空]
   - 页面错误数: 2

4. 处理页面引用3：
    - 页面3不在帧中，发生页面错误。
    - 帧: [1, 2, 3]
    - 页面错误数: 3
5. 处理页面引用2：
    - 页面2已经在帧中，不发生页面错误。
    - 帧: [1, 2, 3]
6. 处理页面引用1：
    - 页面1已经在帧中，不发生页面错误。
    - 帧: [1, 2, 3]
7. 处理页面引用5：
    - 页面5不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面1。
    - 帧: [5, 2, 3]
    - 页面错误数: 4
8. 处理页面引用2：
    - 页面2已经在帧中，不发生页面错误。
    - 帧: [5, 2, 3]
9. 处理页面引用1：
    - 页面1不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面2。
    - 帧: [5, 1, 3]
    - 页面错误数: 5
10. 处理页面引用6：
    - 页面6不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面3。
    - 帧: [5, 1, 6]
    - 页面错误数: 6
11. 处理页面引用2：
    - 页面2不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面5。
    - 帧: [2, 1, 6]
    - 页面错误数: 7
12. 处理页面引用5：
    - 页面5不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面1。

- 帧: [2, 5, 6]
- 页面错误数: 8

13. 处理页面引用6:
    - 页面6已经在帧中，不发生页面错误。
    - 帧: [2, 5, 6]
14. 处理页面引用3:
    - 页面3不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面2。
    - 帧: [3, 5, 6]
    - 页面错误数: 9
15. 处理页面引用1:
    - 页面1不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面5。
    - 帧: [3, 1, 6]
    - 页面错误数: 10
16. 处理页面引用3:
    - 页面3已经在帧中，不发生页面错误。
    - 帧: [3, 1, 6]
17. 处理页面引用6:
    - 页面6已经在帧中，不发生页面错误。
    - 帧: [3, 1, 6]
18. 处理页面引用1:
    - 页面1已经在帧中，不发生页面错误。
    - 帧: [3, 1, 6]
19. 处理页面引用2:
    - 页面2不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面6。
    - 帧: [3, 1, 2]
    - 页面错误数: 11
20. 处理页面引用4:
    - 页面4不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面3。
    - 帧: [4, 1, 2]
    - 页面错误数: 12
21. 处理页面引用3:

- 页面3不在帧中，发生页面错误。根据FIFO算法，移出最早进入的页面1。
- 帧: [4, 3, 2]
- 页面错误数: 13

总结：

FIFO算法在处理给定的页面引用序列时，总共发生了13次页面错误。

3.

3. Consider a disk queue with I/O requests on the following cylinders in their arriving order:

44, 20, 95, 4, 50, 52, 47, 61, 87, 25

We assume a disk with 100 tracks and the disk head is initially located at track 50.

Write the sequence in which requested tracks are serviced using the **Shortest Seek Time First (SSTF) algorithm** and calculate the **total head movement** (in number of cylinders) incurred while servicing these requests. (6 marks)

请求队列和初始状态：

- 请求队列：44, 20, 95, 4, 50, 52, 47, 61, 87, 25
- 初始磁头位置：50

SSTF调度过程：

1. 当前磁头位置：50
   - 最近的请求：50
   - 磁头移动：0
   - 磁头位置更新为：50
   - 更新请求队列：44, 20, 95, 4, 52, 47, 61, 87, 25
2. 当前磁头位置：50
   - 最近的请求：52
   - 磁头移动：2
   - 磁头位置更新为：52
   - 更新请求队列：44, 20, 95, 4, 47, 61, 87, 25

3. 当前磁头位置：52
    - 最近的请求：47
    - 磁头移动：5
    - 磁头位置更新为：47
    - 更新请求队列：44, 20, 95, 4, 61, 87, 25
4. 当前磁头位置：47
    - 最近的请求：44
    - 磁头移动：3
    - 磁头位置更新为：44
    - 更新请求队列：20, 95, 4, 61, 87, 25
5. 当前磁头位置：44
    - 最近的请求：61
    - 磁头移动：17
    - 磁头位置更新为：61
    - 更新请求队列：20, 95, 4, 87, 25
6. 当前磁头位置：61
    - 最近的请求：87
    - 磁头移动：26
    - 磁头位置更新为：87
    - 更新请求队列：20, 95, 4, 25
7. 当前磁头位置：87
    - 最近的请求：95
    - 磁头移动：8
    - 磁头位置更新为：95
    - 更新请求队列：20, 4, 25
8. 当前磁头位置：95
    - 最近的请求：25
    - 磁头移动：70
    - 磁头位置更新为：25
    - 更新请求队列：20, 4
9. 当前磁头位置：25

- 最近的请求：20
- 磊头移动：5
- 磁头位置更新为：20
- 更新请求队列：4

10. 当前磁头位置：20
    - 最近的请求：4
    - 磁头移动：16
    - 磁头位置更新为：4
    - 更新请求队列：空

总磁头移动距离计算：

- 磁头移动：0 + 2 + 5 + 3 + 17 + 26 + 8 + 70 + 5 + 16 = 152

4

4. A paging scheme uses a **Translation Lookaside Buffer** (TLB). A **TLB** access takes 20 ns, and a main memory access takes 50 ns. What is the effective access time (in ns) if the Translation Lookaside buffer TLB hit ratio is 70% and there is no page fault? (6 marks)

计算步骤：

1. TLB命中情况：
    - 当TLB命中时，时间花费为：TLB访问时间 + 主存访问时间
    - 即：20 ns + 50 ns = 70 ns
2. TLB未命中情况：
    - 当TLB未命中时，时间花费为：TLB访问时间 + 2次主存访问时间（一次访问页表，一次访问实际数据）
    - 即：20 ns + 2 × 50 ns = 120 ns
3. 总的有效访问时间（EAT）：
    - 有效访问时间 = TLB命中率 × TLB命中时的时间 + TLB未命中率 × TLB未命中时的时间
    - TLB未命中率 = 1 - TLB命中率 = 1 - 0.7 = 0.3

EAT=0.7×70 ns+0.3×120 nsEAT=0.7×70ns+0.3×120ns

计算EAT：

EAT=0.7×70 ns+0.3×120 nsEAT=0.7×70ns+0.3×120nsEAT=49 ns+36 nsEAT=49ns+36nsEAT=85 nsEAT=85ns

5

5. A program has been divided into five modules. Their lengths and base addresses are stored in the segment table, as depicted in the following space:

Segment number	Length	Base address
0	300	4000
1	600	1000
2	500	2700
3	900	1800
4	1000	2500

What will be the physical memory address for the following logical addresses?

Show the physical memory mapping for the segments. (6 marks)

Segment numbers	Offset d
1	550
3	908
4	670

逻辑地址转换为物理地址：

1. 段1，偏移量550：

- 检查偏移量：550 < 600（合法）
- 物理地址 = 基址 + 偏移量 = 1000 + 550 = 1550

2. 段3，偏移量908：
   - 检查偏移量：908 > 900（非法）
   - 由于偏移量超出段长度，这是一个无效的地址，因此无法转换为物理地址。
3. 段4，偏移量670：
   - 检查偏移量：670 < 1000（合法）
   - 物理地址 = 基址 + 偏移量 = 2500 + 670 = 3170

Q3

QUESTION III. Resource allocation **(12 marks)**

Consider a system with the following information.

R1	R2	R3
1	5	2

Process	Max			Allocation		
	R1	R2	R3	R1	R2	R3
P1	0	0	1	0	0	1
P2	1	7	5	1	0	0
P3	2	3	5	1	3	5
P4	0	6	5	0	6	3

Is this system currently in a safe or unsafe state? Why? Explain. (6 marks)

If a request from P2 arrives for (0, 5, 0), can that request be safely granted immediately? Explain the answer. (6 marks)

可用资源（Available）：

Available=[1,5,2]Available=[1,5,2]

需求矩阵（Need）计算：

Need=Max−AllocationNeed=Max−Allocation

进程	Need R1	Need R2	Need R3
P1	0 - 0 = 0	0 - 0 = 0	1 - 1 = 0
P2	1 - 1 = 0	7 - 0 = 7	5 - 0 = 5
P3	2 - 1 = 1	3 - 3 = 0	5 - 5 = 0
P4	0 - 0 = 0	6 - 6 = 0	5 - 3 = 2

进程	Need R1	Need R2	Need R3
P1	0	0	0
P2	0	7	5
P3	1	0	0
P4	0	0	2

我们将尝试找到一个安全序列：

1. 检查P1：Need ≤ Available，满足条件。
    - 分配P1后，释放其资源：[0, 0, 1]
    - 更新Available：[1+0, 5+0, 2+1] = [1, 5, 3]
    - 安全序列：P1
2. 检查P2：Need ≤ Available，不满足条件。
3. 检查P3：Need ≤ Available，满足条件。
    - 分配P3后，释放其资源：[1, 3, 5]
    - 更新Available：[1+1, 5+3, 3+5] = [2, 8, 8]
    - 安全序列：P1, P3
4. 检查P2：Need ≤ Available，满足条件。
    - 分配P2后，释放其资源：[1, 0, 0]
    - 更新Available：[2+1, 8+0, 8+0] = [3, 8, 8]
    - 安全序列：P1, P3, P2
5. 检查P4：Need ≤ Available，满足条件。
    - 分配P4后，释放其资源：[0, 6, 3]
    - 更新Available：[3+0, 8+6, 8+3] = [3, 14, 11]
    - 安全序列：P1, P3, P2, P4

由于找到了安全序列，系统当前处于安全状态。

6. 如果P2请求（0, 5, 0），请求能否被立即安全地授予？

首先，检查请求是否小于等于可用资源：

Request=[0,5,0]Request=[0,5,0]Available=[1,5,2]Available=[1,5,2]

显然，Request ⩽ Available。

假设分配了请求：

- 更新分配后的Available资源：
  New Available=Available－Request=[1－0,5－5,2－0]=[1,0,2]New Available=Available－Request=[1－0,5－5,2－0]=[1,0,2]
- 更新P2的Allocation：
  New Allocation P2=Old Allocation P2+Request=[1+0,0+5,0+0]=[1,5,0]New Allocation P2=Old Allocation P2+Request=[1+0,0+5,0+0]=[1,5,0]
- 更新P2的Need：
  New Need P2=Old Need P2－Request=[0－0,7－5,5－0]=[0,2,5]New Need P2=Old Need P2－Request=[0－0,7－5,5－0]=[0,2,5]

新的分配情况：

进程	Max	Allocation	Need
P1	[0, 0, 1]	[0, 0, 1]	[0, 0, 0]
P2	[1, 7, 5]	[1, 5, 0]	[0, 2, 5]
P3	[2, 3, 5]	[1, 3, 5]	[1, 0, 0]
P4	[0, 6, 5]	[0, 6, 3]	[0, 0, 2]

可用资源：[1, 0, 2]

验证系统是否安全：

1. 检查P1：Need ⩽ Available，满足条件。
   - 分配P1后，释放其资源：[0, 0, 1]
   - 更新Available：[1+0, 0+0, 2+1] = [1, 0, 3]
   - 安全序列：P1
2. 检查P2：Need ⩽ Available，不满足条件。
3. 检查P3：Need ⩽ Available，满足条件。
   - 分配P3后，释放其资源：[1, 3, 5]
   - 更新Available：[1+1, 0+3, 3+5] = [2, 3, 8]
   - 安全序列：P1, P3
4. 检查P2：Need ⩽ Available，满足条件。

- 分配P2后，释放其资源：[1, 5, 0]
- 更新Available：[2+1, 3+5, 8+0] = [3, 8, 8]
- 安全序列：P1, P3, P2

5. 检查P4：Need ≤ Available，满足条件。
   - 分配P4后，释放其资源：[0, 6, 3]
   - 更新Available：[3+0, 8+6, 8+3] = [3, 14, 11]
   - 安全序列：P1, P3, P2, P4

由于找到了安全序列，P2的请求（0, 5, 0）可以被安全地立即授予。

Q4

QUESTION IV. Operating System in C Language (12 marks)

The following code implements a **simple scenario of a barbershop**, which includes a barber process and multiple customer processes:

1) If there is an empty chair, a customer can sit down and wait for the barber to cut their hair.
2) If there are no empty chairs, the customer leaves.
3) When the barber wakes up, if there are customers waiting, the barber prepares to cut their hair.

```
int waiting = 0;
int CHAIRS = N;
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
```


barber process:	consumer process:

process barber(){
  while(true){
  P(customers);
  P(mutex);
  waiting--;
  V(barbers);
  V(mutex);
  cut_hair(); //cutting hair
  }
}
	process customer_i(){
	  P(mutex);
	  if(waiting < CHAIRS){
	  waiting++;
	  V(customers);
	  V(mutex);
	  P(barbers);
	  get_haircut(); //receiving service
	  }else{
	  V(mutex);
	  }
	}

(1) What semaphores are used in this code? What are the implication of the initial value of each semaphore? (4 marks)

(2) What would happen if "waiting--" and "waiting++" were placed after V(mutex)? (4 marks)

(3) What would happen if the value of CHAIRS is set to 0 in this code? (4 marks)

1. 代码中使用的信号量及其初始值的含义

代码中使用了以下信号量：

- `customers`：初始值为0，用于表示等待理发的顾客数量。当有顾客到来时，此信号量增加；当理发师准备理发时，此信号量减少。
- `barbers`：初始值为0，用于表示理发师的可用状态。当理发师准备理发时，此信号量增加；当理发师开始理发时，此信号量减少。
- `mutex`：初始值为1，用于保护对共享变量 `waiting` 的访问，确保在任何时刻只有一个进程能够修改该变量，从而避免竞争条件。

2. 如果将“`waiting--`”和“`waiting++`”放在“`V(mutex)`”之后会发生什么？

如果将“`waiting--`”和“`waiting++`”放在“`V(mutex)`”之后，会出现竞态条件。具体情况如下：

- 当一个顾客进来并发现有空椅子时，它将进入临界区并增加 `waiting`，但如果在它增加之前被中断，另一个顾客也会进入临界区并检查 `waiting` 值。
- 类似地，当理发师在减少 `waiting` 时，如果在它减少之前被中断，另一个顾客可能会进来并更新 `waiting` 值。

这种情况下，多个顾客可能会同时修改 `waiting` 值，导致错误的值出现，影响系统的正确性。

3. 如果将 `CHAIRS` 的值设为0，会发生什么？

如果将 `CHAIRS` 的值设为0，表示理发店没有等待椅子。具体影响如下：

- 只有当理发师空闲时，顾客才能立即接受服务。因为没有等待椅子，如果理发师正在服务其他顾客，新到的顾客会直接离开。
- 在代码中，顾客进来后检查 `waiting < CHAIRS` 的条件将始终为假，因为 `CHAIRS` 为0。因此，所有新来的顾客在 `else` 分支中直接执行 `V(mutex)` 并离开。