Worksheet 6.1: Basic CPU Scheduling

(a) Schedule the following input using **Preemptive Priority Scheduling**. Assume that a smaller number indicates higher priority. First give the output in the form of a **Gantt chart**, and then compute the **average waiting time**. Show your work.

Process	Arrival Time	CPU Burst	Priority
P_1	0	5	3
P_2	1	3	4
P_3	3	2	1
P_4	6	4	2

.
Gantt Chart:

Time:	0	3	5	6	10	11
Process:	P_1	P_3	P_1	P_4	P_1	P_2

P_1 waiting time = 2+4 = 6

P_2 waiting time = 11-1 = 10

P_3 waiting time = 3-3 = 0

P_4 waiting time = 6-6 = 0

Average waiting time = (6+10) / 4 = 4

(b) Schedule the following input using **Round Robin Scheduling** with a time quantum of **5**. Assume that all processes arrive at time 0 but assign time quanta in the given order (P_1, P_2, P_3, P_4). First give the output in the form of a **Gantt chart**, and then compute the **average waiting time**. Show your work.

Process	CPU Burst
P_1	8
P_2	3
P_3	12
P_4	6

Gantt Chart:

Time:	0	5	8	13	18	21	26	27
Process:	P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_3

P_1 waiting time = 18-5 = 13

P_2 waiting time = 5

P_3 waiting time = 8 + (21-13) + 1 = 8 + 8 + 1 =17

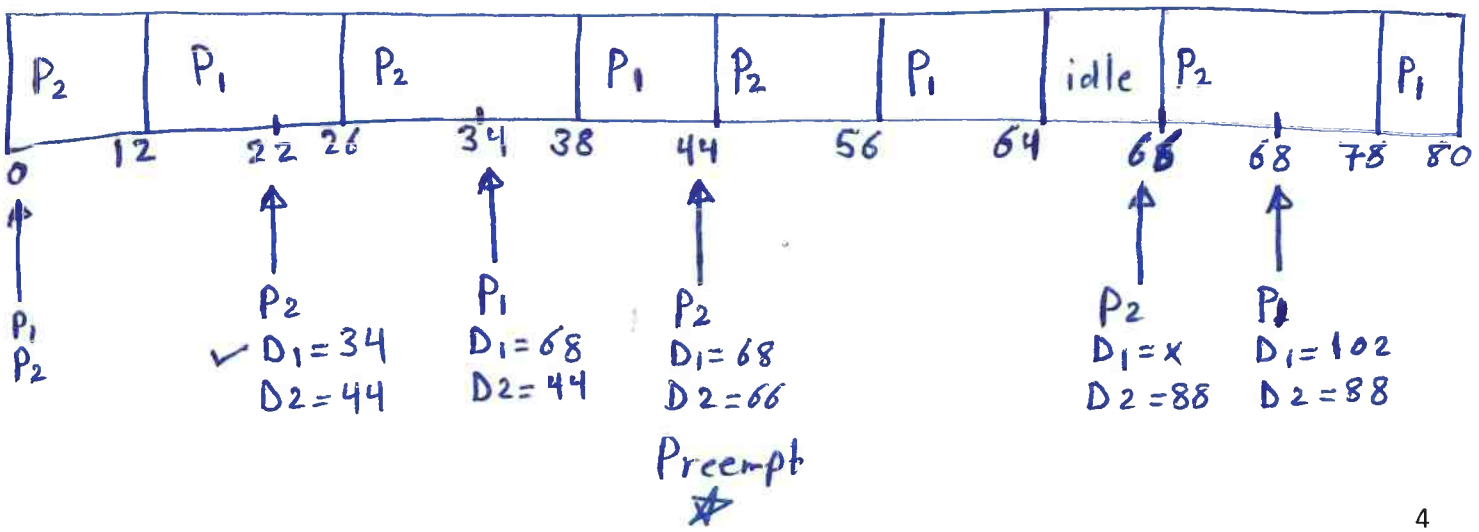
P_4 waiting time = 13 + 26 – 18 = 13 + 8 = 21

Average waiting time = (13 + 5 + 17 + 21) / 4 = 56/4 = 14

(b) Schedule the following two **periodic** processes using **Earliest-Deadline-First Scheduling**. Give the output in the form of a **Gantt chart**. Justify the scheduling decision made at the arrival of each new CPU burst by **giving the deadlines** at that point. If at some point, both processes have the same deadline, resolve the tie in favor of the process that currently has the CPU. Stop your scheduling as soon as a process misses its deadline or when you reach Time **80**, whichever occurs first. Clearly indicate if a process misses its deadline.

P_1: $p_1=34$, $t_1=14$, $d_1=34$
P_2: $p_2=22$, $t_2=12$, $d_2=22$

Recall that p is the period, t is the length of the CPU burst, and d is the deadline. So, P_1 will have CPU bursts of length 14 periodically arriving at times 0, 34, 68, …, and each P_1 burst must complete executing before the arrival of the next P_1 burst. (10 points)



Worksheet 6.3: CPU Scheduling, Multiple-Choice Questions

Choose the right answer. There is only **one** correct answer.

1. Which of the following is (are) true about the difference between the Round Robin (RR) and the Shortest-Job-First (SJF) scheduling algorithms?
   a. RR minimizes the average waiting time, but SJF does not.
   b. SJF minimizes the average waiting time, but RR does not.
   c. RR may cause starvation, but SJF does not cause starvation.
   d. SJF may cause starvation, but RR does not cause starvation.
   e. Both a and d are correct **f. Both b and d are correct.** g. Both b and c are correct.

2. Which of the following is true about multilevel-feedback-queue scheduling?
   a. If a process uses its entire time quantum, it is moved to a higher priority level.
   b. If a process uses its entire time quantum, it is moved to a lower priority level.
   c. If a process spends a lot of time at a low-priority level without getting the CPU, it is moved to a higher priority level.
   d. If a process spends a lot of time at a high-priority level without getting the CPU, it is moved to a lower priority level.
   e. Both b and d are true. f. Both a and c are true. **g. Both b and c are true**

3. What's the difference between Rate-Monotonic Scheduling (RM) and Earliest-Deadline-First (EDF) Scheduling?
   a. EDF is preemptive, while RM is not.
   b. EDF assigns priorities while RM does not.
   c. EDF is more likely to meet the deadlines.
   d. EDF uses fixed priorities while RM dynamically adjusts priorities.
   e. Both c and d are correct. f. Both a and c are correct. g. Both a and d are correct.

4. How do **processor affinity** and **load balancing** interact in a multi-processor environment?
   a. Satisfying processor affinity always makes the load less balanced.
   b. Processor affinity and load balancing always conflict with each other.
   c. Improving load balancing may conflict with the processor affinity requirement in some cases.
   d. Processor affinity and load balancing are totally unrelated and can be handled independently.
   e. Improving load balancing always satisfies the processor affinity requirement.

CSC 139: Operating Systems Principles
Second Quiz, Fall 2020
Friday, April 17th, 2020
Section 4
Instructor: Dr. Ghassan Shobaki

Student Name: Key **Student Number:** _____

Q1. Answer with TRUE or FALSE. **[40 points]**

1. If in the **avoidance** version of the **Banker's algorithm**, we fail to find a **safe sequence**, the system **will necessarily** have a deadlock. TRUE (FALSE)

2. If there are multiple processes that are holding some resources and waiting for other resources, we necessarily have a deadlock. TRUE (FALSE)

3. A deadlock cannot happen unless mutual exclusion is required for some resources. (TRUE) FALSE

4. In theory, it is possible to recover from a deadlock by preempting some processes from some resources **without** terminating any process. (TRUE) FALSE

Q2. Circle the right answer. There is only **one** correct answer. **[60 points]**

1. The worst-case running time for finding a safe sequence in the **Banker's algorithm** for deadlock detection on a system with p processes and q resource types is:
   a. $O(p^2)$ b. $O(q^2)$ (c.) $O(p^2q)$ d. $O(pq^2)$ e. $O(pq)$ f. $O(p^2q^2)$

2. Which of the following statements is (are) true about cycles in the Resource Allocation Graph (RAG)?
   ✓a. Having a cycle in the RAG is always a necessary condition for deadlocks.
   b. Having a cycle in the RAG is always a sufficient condition for deadlocks.
   c. Having a cycle in the RAG necessarily implies a deadlock if all resource types have multiple instances
   ✓d. Having a cycle in the RAG necessarily implies a deadlock if all resource types have single instances.
   e. Both a and c are true. (f.) Both a and d are true. g. a, b and d are true.

3. Which of the following is a **necessary** condition for deadlocks:
   a. Some resources are preemptive b. All resources are preemptive
   (c.) Some resources are non-preemptive d. All resources are non-preemptive
   e. Both a and b are true. f. Both c and d are true.

4. Given the following state of a system that has one resource type:

	Current Allocation	Current Request
P_0	2	4
P_1	3	8
P_2	1	2
P_3	1	6

Avail: 2 → 3 → 4; P_2: 3 → 4; P_0: 6; P_3: 7 → 8

What's the minimum value for the **total** number of instances that will make the system deadlock free?
a. 7 b. 8 c. 9 d. 10 (e.) 11 f. 12 g. 13 h. 14

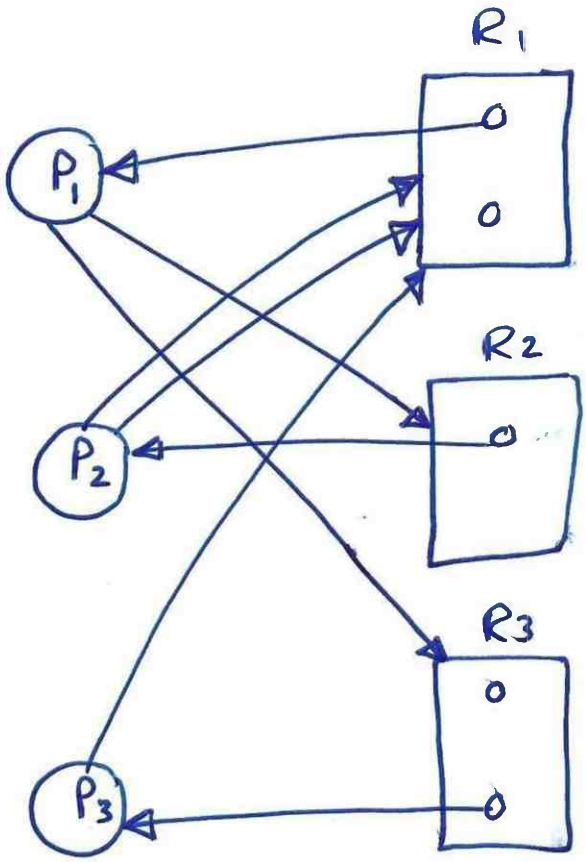
$4 + (2 + 3 + 1 + 1) = 11$

5. Consider a system with processes P_1, P_2 and P_3 and resource types R_1, R_2 and R_3. There is **one** instance of R_2 and there are **two** instances of each of R_1 and R_3. P_1 is currently holding an instance of R_1 and requesting an instance of R_2 and an instance of R_3, P_2 is holding an instance of R_2 and requesting two instances of R_1, and P_3 is holding an instance of R_3 and requesting an instance of R_1. What's the current state of the system?
   a. P_2 and P_3 are in a deadlock but P_1 is not in a deadlock.
   (b.) P_1 and P_2 are in a deadlock but P_3 is not in a deadlock.
   c. P_1 and P_3 are in a deadlock but P_2 is not in a deadlock.
   d. All three processes are in a deadlock. e. There is no deadlock in the system.

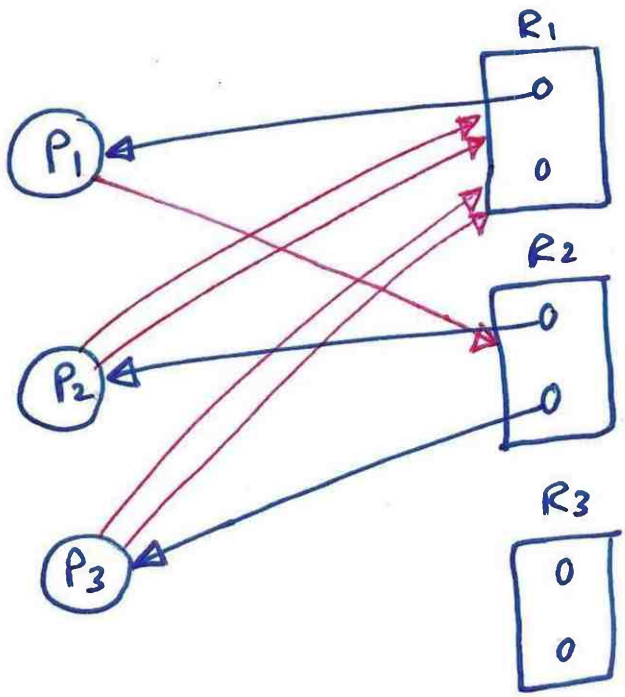
6. Consider a system with processes P_1, P_2 and P_3 and resource types R_1 and R_2. There are **two** instances of each resource type. If P_1 is currently holding an instance of R_1, P_2 is holding an instance of R_2, and P_3 is holding an instance of R_2, which of the following sequences of requests will **necessarily** cause a deadlock?

a. P_1 requests one instance of R_2, P_2 requests one instance of R_1 and P_3 requests one instance of R_1
b. P_1 requests one instance of R_2, P_2 requests one instance of R_1 and P_3 requests two instances of R_1
c. P_1 requests one instance of R_2, P_3 requests one instance of R_1 and P_2 requests two instances of R_1
d. P_1 requests one instance of R_2,P_2 requests two instances of R_1 and P_3 requests two instances of R_1
e. b and d are correct
f. c and d are correct
g. b, c and d are correct

Q 5



Q 6



Worksheet 7.1: Deadlocks, Problem Solving

Q1. Given the following snapshot of a system, answer the following questions using the **avoidance** version of the **Banker's Algorithm**:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	3 1 1	6 3 6	1 2 0	3 2 5
P_1	0 0 3	2 1 4		2 1 1
P_2	0 0 1	6 4 6		6 4 5
P_3	2 0 1	3 1 1		1 1 0
P_4	1 1 1	3 2 5		2 1 4

1. Show that the system is in a safe state by finding a safe sequence. Show your work.

	Work
	1 2 0
P_3	3 2 1
P_1	3 2 4
P_4	4 3 5
P_0	7 4 6
P_2	7 4 7

2. If a request from Process P_0 arrives for (0, 1, 0), can this request be granted immediately? Show your work.
   Allocation of P_0 becomes 3 2 1
   Need of P_0 becomes 3 1 5
   Avail becomes 1 1 0

	Work
	1 1 0
P_3	3 1 1
P_1	3 1 4
P_4	4 2 5
P_0	7 4 6
P_2	7 4 7

Since the request will put the system in a safe state, the request will be granted.

Q2. Given the following snapshot of a system, answer the following questions using the **<u>detection</u>** version of the **Banker's Algorithm**:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	2	2	1	1	2	2	1	3
P_1	2	1	1	1	4	5			
P_2	1	1	0	3	3	5			
P_3	1	2	1	2	2	6			
P_4	3	0	0	1	2	1			

1. Show that the system is in a safe state by finding a safe sequence. When searching for a process whose request can be satisfied, check the processes in the given order, as we were doing in class. Show your work.

	Avail		
	2	1	3
P_0	2	3	5
P_4	5	3	5
P_2	6	4	5
P_1	8	5	6
P_3	9	7	7

2. How many safe sequences are there for the above snapshot? Briefly justify your answer.

   Only 1. At each step in the construction of the safe sequence, there is only one choice.

3. If P_1 requests an additional instance of each of A and C, that is, the request vector for P_1 becomes (2, 4, 6), will this request result in a deadlock? If the answer is no, give a safe state. If the answer is yes, identify the processes that will be involved in the deadlock. Remember that this is the **<u>detection</u>** version of the algorithm. Show your work.

	Avail		
	2	1	3
P_0	2	3	5
P_4	5	3	5
P_2	**6**	**4**	**5**

   Does not satisfy the need of P_1 or P_3
   P_1 and P_3 will be in a deadlock

Worksheet 5.4: The Dining-Philosophers Problem

Consider the following monitor solution to the **Dining-Philosophers Problem**.

```
monitor DiningPhilosophers {
        enum { THINKING; HUNGRY, EATING) } state [5] ;
        condition self [5];

   void pickup (int i) {

1               state[i] = HUNGRY;
2               test(i);
3               if (state[i] != EATING) self[i].wait;
}
            void putdown (int i) {
1               state[i] = THINKING;
2               test((i+1)%5);
3               test((i-1)%5);
}
```

```
1   void test (int i) {
2    printf("\nTesting Phil. %d", i);
3           if ((state[(i + 1) % 5] != EATING) &&
4           (state[i] == HUNGRY) &&
5           (state[(i -1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal ();
    printf("\nPhils. %d and %d not eating", (i+1)%5, (i-1)%5);
    printf("\nPhil. %d can eat", i);
            }
            else {
    printf("\nPhil. %d cannot eat", i);
            }
    }
```

1. Considering the print statements added to the test function, show the output printed by the above code if the following sequence of operations are executed.
   pickup (2);
   pickup (1);
   putdown (2);

   Testing Phil. 2
   Phils. 3 and 1 not eating
   Phil. 2 can eat

   Testing Phil. 1
   Phil. 1 cannot eat

   Testing Phil. 3
   Phil. 3 cannot eat

   Testing Phil. 1
   Phils. 2 and 0 not eating
   Phil. 1 can eat

2. Why does the above solution prevent deadlocks? First describe the scenario that causes a deadlock and then explain why this scenario is impossible with the above monitor solution. (**Limit: 3 lines)**

   Deadlock (DL) happens if **each** phil. picks one chopstick and waits for the other. DL cannot happen here, because a monitor function is a critical section (only one process can be in the monitor at any time). So, a phil. checks both chopsticks in a critical section and eats only if both are available.

3. Which lines in the **test()** method are not needed when **test()** is called from **pickup()**? Explain why each of these lines is not needed. (**Limit: 2 lines each)**

   Line 2 is not needed, because the state of phil. i is already set to HUNGRY in pickup()

   Line 5 is not needed, because phil. i is already active in the monitor and is not waiting on a condition variable. So, there is no need to signal it.

4. The problem solved by the above monitor models 5 processes (philosophers) sharing 5 resources (chopsticks), where each Process i shares a resource with Process (i+1) mod 5 and Process (i-1) mod5. Suppose that we modify the problem to be as follows:
   There are n processes (philosophers) sharing n resources (chopsticks), where each Process i shares k resources with k other processes (k<n). More specifically, Process i shares a resource with Process (i+1) mod n, a second resource with Process (i+2) mod n, …. and a kth resource with Process (i+k) mod n. Each Process i cannot operate (eat) unless it has exclusive access to all **k** resources. Modify the above code to solve this problem. You only need to rewrite the functions that change after identifying them. **Ignore the print statements in this part**.

```
void putdown (int i) {
      if (state[i] != EATING) exit (1);
      state[i] = THINKING;
        for (j= 1; j <= k ; j++)
            test ((i + j) % n);
}

void test (int i) {

       if (state[i] != HUNGRY) return;
      for (j=1; j<=k; j++)
           if (state[(i + j) % n] == EATING)
                return;

       state[i] = EATING ;
       self[i].signal ();
}
```

Work Sheet 5.1: HW Solutions to the Critical-Section Problem

Consider the following solution to the **Critical Section Problem** using the **test_and_set** hardware instruction.

```
for (iter = 1; iter<=2; iter++) {
1    waiting[i] = true;
2    key = true;
     printf("\n Process %d is waiting", i);
3    while (waiting[i] && key)
4        key = test_and_set(&lock);
5    waiting[i] = false;

printf("\n Process %d enters CS", i);

     /* critical section */

6   j = (i + 1) % n;
7   while ((j != i) && !waiting[j])
8        j = (j + 1) % n;
printf("\n j = %d", j);
9    if (j == i)
10     lock = false;
11  else
12     waiting[j] = false;

   /* remainder section */

}
```

1. Assume that there are six processes (P_0, P_1, …, P_5). Considering the print statements added to the above code, show the output printed by the code if the following sequence of events takes place (the order is very important):
   P_5 executes Line 3 in its first iteration
   P_4 executes Line 3 in its first iteration
   P_0 executes Line 3 in its first iteration

   Assume that the remainder section is so short that if P_x exits its critical section and P_y enters its critical section next, P_x will complete its remainder section and execute Line 3 in its second iteration before P_y exits its critical section. So, when P_y exits its critical section, P_x will be waiting to enter the critical section for the second time.

   Note: Parenthesized text is for explanation only (it does not actually appear in the output)

   Process 5 is waiting

   Process 5 enters CS
   Process 4 is waiting
   Process 0 is waiting
   j = 0 (because the loop on Lines 7 and 8 searches in the order 0, 1, 2, 3, 4, 5)

   Process 0 enters CS
   Process 5 is waiting (for its second turn)
   j = 4 (because the loop on Lines 7 and 8 searches in the order 1, 2, 3, 4, 5, 0)

Process 4 enters CS
Process 0 is waiting (for its second turn)
j = 5 (because the loop on Lines 7 and 8 searches in the order 5, 0, 1, 2, 3, 4)

Process 5 enters CS (for the second time)
Process 4 is waiting (for its second turn)
j = 0

Process 0 enters CS (for the second time)
j = 4

Process 4 enters CS (for the second time)
j = 4 (no processes are waiting)

2. Can the above solution cause starvation? If yes, give a scenario (sequence of events) that causes starvation. If not, explain why. Of course, you must answer this question for the general case where each process may request access to the critical section an arbitrary number of times, not only two times. (**Limit: 3 lines**).

   Of course, it won't, because when a process is done with the CS, it checks to see if there are waiting processes and will give the CS to one of them. So, a process won't take the CS for a second time unless no other process is waiting.

3. If the total number of processes is **n**, what's the maximum number of other processes that a waiting process may wait for before entering the critical section?
   n-1

Worksheet 8.1: Main Memory

Q1. Choose the right answer. There is only **<u>one</u>** correct answer.

1. What's the relationship between page size and fragmentation?
   a. A larger page size decreases internal fragmentation.
   b. A larger page size increases internal fragmentation.
   c. A larger page size decreases external fragmentation.
   d. A larger page size increases external fragmentation.
   e. Both a and c are correct. f. Both b and d are correct. g. None of the above is correct.

2. A logical address space of a process has 512 pages with an 8-KB page size. How many bits are needed in the logical address?
   a. 8 b. 12 c. 16 d. 20 **e. 22** f. 24 g. 40 h. 64

   $512 = 2^9$ -> 9 bits for the base
   $8K = 2^3 \times 2^{10} = 2^{13}$ -> 13 bits for the offset
   Total number of bits = 9 + 13 = 22 bits

3. If 32 bits are used to represent a logical address, and the page size is 16 KB, what's the maximum number of pages that a process can have in its logical address space?
   a. 1M b. 512K **c. 256K** d. 128K e. 64K f. 32K g. 16K h. 8K i. 4K

   $16K = 2^{14}$ -> 14 bits for offset
   Bits left for the base = 32 – 14 = 18 bits
   $2^{18} = 256$ K

4. If a process uses 1100B of memory in a system with a page size of 512B, what's the size of internal fragmentation?
   a. 24B b. 76B c. 100B d. 412B **e. 436B** f. 0B

   What's the minimum number of pages needed?
   2 pages: 2 x 512 = 1024 not enough
   3 pages: 3 x 512 = 1536 enough

   Internal fragmentation = 1536 – 1100 = 436 B

5. Which of the following is (are) true about paging and segmentation?
   a. **Paging divides memory into equal blocks, but segmentation may divide it into unequal blocks**.
   b. Segmentation divides memory into equal blocks, but paging may divide it into unequal blocks.
   c. Paging requires hardware support but segmentation does not.
   d. Segmentation requires hardware support but paging does not.
   e. In both segmentation and paging, the address space of a process must be contiguous.
   f. Both a and c are correct. g. Both a and e are correct. h. Both d and e are correct.

6. Which of the following is (are) true about static linking and dynamic linking?
   a. Static linking results in a larger executable.
   b. Dynamic linking results in a larger executable.
   c. Static linking makes it possible for multiple processes to share libraries at run time.
   d. Dynamic linking makes it possible for multiple processes to share libraries at run time.
   e. Both a and c are true. **f. Both a and d are true**. g. Both c and d are true.

Q2. A system has a Table-Lookaside Buffer (TLB) with a negligibly small access time compared to the memory access time. Calculate the TLB hit ratio that will keep the Effective Access Time (EAT) within 10% of the ideal EAT (ideal EAT is EAT with no TLB misses). Show your work **clearly**.

Ideally, all accesses will hit in the TLB. So, there will be one TLB access and one memory access
Let memory-access time be M
TLB access is negligible

EAT = hM + (1-h) 2M

1.1 M = hM + 2 (1-h) M

1.1 = h + 2 (1-h)

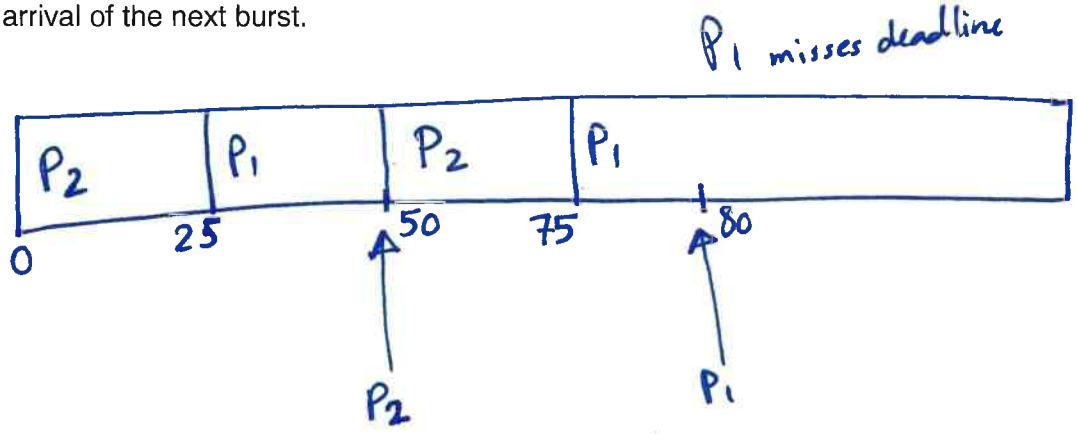
Solve for h:

1.1 = h + 2 -2h
h = 2 – 1.1 = 0.9

(b) Schedule the following two **periodic** processes using **Rate Monotonic Scheduling**. Give the output in the form of a **Gantt chart**. Write a comment explaining the scheduling decision made at the arrival of every new CPU burst. Stop your scheduling as soon as a process misses its deadline or when you reach Time 100, whichever occurs first. **Clearly indicate if a process misses its deadline**. (10 points)

P_1: p_1=80, t_1=35, d_1=80
P_2: p_2=50, t_2=25, d_2=50

Recall that p is the period, t is the length of the CPU burst, and d is the deadline. So, P_1 will have CPU bursts of length 35 periodically arriving at times 0, 80, 160, …, and each burst must be completed before the arrival of the next burst.



Worksheet 5.5: Questions on Assignment 1

Consider the following functions that implement the producer and the consumer in Assignment 1. Assuming that the rest of the code is the same as in the give templates, answer the following questions. Each question is **independent** of other questions.

```
void Producer(int bufSize, int itemCnt, int randSeed){
        int i, in = 0, out = 0, val;

1        for (i=0; i<itemCnt; i++) {
2                while((GetIn()+1)%bufSize == GetOut());
3                val = GetRand(0, 1000);
4                WriteAtBufIndex(in, val);
5                in = (in + 1)%bufSize;
6                SetIn(in);
        }
}
void Consumer(){
//Code to open shared memory block and map it to gShmPtr
1    int bufSize = GetBufSize();
2    int itemCnt = GetItemCnt();
3    int in = GetIn();
4    int out = GetOut();
5    for(i=0; i<itemCnt; i++){
6          while(GetIn() == GetOut());
7          val = ReadAtBufIndex(out);
8          out = (out + 1)%bufSize;
9          SetOut(out);
    }
```

1. Which line in the above code does each of the following? Make sure you specify whether that line is in the Producer or in the Consumer

   Waiting if the buffer is empty — Line 6 in Consumer

   Updating the value of *in* in the shared memory buffer — Line 6 in Producer

2. Assuming 4 items and a buffer size of 3, which of the following outputs are correct and which are incorrect? See the prints in the above code. Circle Right or Wrong.

Prod 0	Prod 0	Prod 0	Prod 0
Cons 0	Prod 1	Prod 1	Prod 1
Prod 1	Cons 0	Cons 0	Cons 0
Prod 2	Prod 2	Prod 2	Cons 1
Cons 1	Cons 1	Prod 3	Prod 2
Prod 3	Prod 3	Cons 1	Cons 2
Cons 2	Cons 2	Cons 2	Cons 3
Cons 3	Cons 3	Cons 3	Prod 3
Right Wrong	**Right** Wrong	Right **Wrong**	Right **Wrong**

3. Use the table below to trace the consequences of making the following changes**,** assuming that everything else remains the same. Assume that there is a **single** producer and a **single** consumer. Let the buffer size be **m**, the number of items be **n**, **n** > **m**. **Inconsistent** answers will get zero credit.

Change	Will it work correctly?	Items Produced	Items Consumed	Explanation
Delete ***SetIn()*** on Line 6 of the Producer	No	n	0	in won't get updated in shared mem. So, cons. would think that nothing has been produced, and would thus consume nothing. Prod. would forget that it has produced an item, and would thus keep producing items but it will overwrite some items.

Change	Will it work correctly?	Items Produced	Items Consumed	Explanation
Delete ***SetIn()*** on Line 6 of the Producer and change ***GetIn()*** on Line 2 of Producer to in.	No	m-1	0	in won't get updated in shared mem. So, cons. would think that nothing has been produced, and would thus consume nothing. Prod. would keep producing until it fills the buf with m-1 items. Then prod. and cons. will be waiting for each other forever.

4. What's the range of possible values that may be returned by **GetIn()** on Line 3 and **GetOut()** on Line 4 of the **Consumer**? Assume a single producer and a single consumer. Let the buffer size be **m**, the number of items be **n**, **n** > **m**.

   Range of all possible values returned by GetIn() on Line 3 in Consumer: 0 to m-1

   Range of all possible values returned by GetOut() on Line 4 in Consumer: 0

5. Suppose that we modify the problem such that there is one producer and multiple consumers with the following specifications:
   - The parent process forks multiple child process, each child running the same **Consumer** code.
   - The producer (in the parent process) produces itemCnt items, and an item may be consumed by any consumer. An item can be consumed **only once** (once one consumer consumes it, the buffer cell may be used by the producer to produce new items).
   - After reading an item, a consumer spends a significant amount of time processing the item. So, the consumer code will have a function named ProcessItem() somewhere after ReadAtBufIndex() (you need to figure out the best place for it). ProcessItem takes significant time but does independent work that each consumer can do in parallel with other consumers.

- When all produced items have been consumed, each consumer should detect this (you will need to figure out a mechanism for doing this) and terminate itself by calling exit(0).

  Modify the above code to solve this single-producer multiple-consumer problem. Don't worry about the parent code that forks multiple child processes. Only give the necessary changes to the Consumer and Producer functions (if any) and describe any changes that need to be made to the header of the shared memory block. Your code will be graded on both correctness and efficiency. Correct but inefficient code will get only partial credit.

Key idea: Need to add to the shared memory header a fifth field that tracks the number of items consumed. Since multiple processes will be updating this shared variable as well as the *out* variable, we must protect access to these variables using a lock or a semaphore.

Let the name of that shared variable be ConsumedCnt
Assume that we have a SetConsumedCnt() and a GetConsumedCnt() functions.

Let sem be a mutex semaphore to protect access to the shared variables. sem must be initialized to 1.

```
while(1){
            sem_wait(&sem);
            if (GetConsumedCnt() == itemCnt)
            { sem_post(&sem); exit(0);}
            out = GetOut();
            while(GetIn() == out);
            val = ReadAtBufIndex(out);
            SetConsumedCnt(GetConsumedCnt() + 1);
            out = (out + 1) % bufSize;
            SetOut(out);
            sem_post(sem);
            ProcessItem(val);
}
```

Worksheet 5.6: Assignment 2

Consider the following code that implements the multithreaded program of Assignment 2:

```
int gData[MAX_SIZE]; //Array that holds the data
int gThreadCount; // Number of threads
int gDoneThreadCount; // Number of threads that are done so far
int gThreadProd[MAX_THREADS]; // The product of each thread
sem_t sem_1, sem_2;
#define NUM_LIMIT 9973

int main(int argc, char *argv[]){
1      pthread_t tid[MAX_THREADS];
2      pthread_attr_t attr[MAX_THREADS];
3      int indices[MAX_THREADS][3], i, arraySize, prod;

// Code for parsing command-line arguments and generating input

4    CalculateIndices (arraySize, gThreadCount, indices);
5    InitSharedVars ();
6    sem_init (&sem_1, 0, X1);
7    sem_init (&sem_2, 0, X2);
8    for (i=0; i<gThreadCount; i++)
9           pthread_create(tid+i, attr+i, Child, indices[i]);

10   sem_wait(X3);
11   prod = ComputeTotalProduct();
12   for (i=0; i<gThreadCount; i++)
13          pthread_cancel(tid[i]);
14   printf ("Product is %d\n", prod);
}

void* Child (void *param) {
1           int threadNum = ((int*)param)[0];
2           int start = ((int*)param)[1];
3           int end = ((int*)param)[2];

4           gThreadProd[threadNum] = 1;
5           for (int i=start; i<=end; i++) {
6               if (gData[i] == 0) {
7                       gThreadProd[threadNum] = 0;
8                       gThreadDone[threadNum] = true;
9                       sem_post(&sem_2);
10                      pthread_exit(0);
                }
11              gThreadProd[threadNum] *= gData[i];
12              gThreadProd[threadNum] %= NUM_LIMIT;
            }
13          gThreadDone[threadNum] = true;

14          sem_wait(&sem_1);
15          gDoneThreadCount++;
16          if (gDoneThreadCount == gThreadCount)
17               sem_post(&sem_2);
18          sem_post(&sem_1);

}
```

1. Replace X1, X2, X3 in the above code with the right values or expressions.

   X1: 1

   X2: 0

   X3: &sem_2

2. Trace the consequences of changing sem_2 on Line 9 of the child to sem_1. Clearly indicate if the program will work correctly or not.

   If there is a zero in the input, the parent will never be notified and the program will not terminate. Multiple children could be updating shared variable gDoneThreadCount at the same time, which means that the shared variable may have an incorrect value.

3. Suppose that you have two separate programs for solving the above problem. One program does sequential search without dividing the array and another program divides the array into equal divisions and creates a thread to process each division **as in the above code**. The sequential program terminates as soon as a zero is found (if there is a zero).
   The search part of the sequential program takes 80% of the program's execution time if the input array has no zero. The other 20% is spent in parsing the command-line arguments and generating the input data, which is something that both programs do sequentially. Given that the input array is divided into 8 divisions, compute the speedup achieved by the multithreaded program relative to the sequential program in each of the two cases below. Assume **ideal parallelism** on a system with at least 9 cores. Ignore all kinds of overhead, and assume that all threads start immediately.

   First Case: The parent waits for all children (no semaphores used), and there is **no zero** in the input.

   Speedup = sequential time / parallel time = 1 / (0.2 + 0.8/8) = 1 / (0.2 + 0.1) = 10 / 3

   Second Case: Semaphores are used as in the above code, and there is a zero right at the beginning of the 8th division.

   Speedup = sequential time / parallel time = (0.2 + 0.8 x 7/8) / (0.2 + 0) = 0.9 / 0.2 = 4.5

4. If we replace Lines 8-13 in the Parent with the following:

```
for(i=0; i<gThreadCount; i++)
{
      pthread_create(tid+i, attr+i, Child, indices[i]);
      pthread_join(tid+i);
}
prod = ComputeTotalProduct();
```

   1. Will the program still compute the product correctly? **YES** NO

   2. Assuming that there are no zeros in the input, will the execution time in this case be the same as Method 1, Method 2, Method 3 or as the sequential algorithm? First circle the right answer, and then Justify your answer.

      Sequential: Don't divide the array and just scan it sequentially

      Method 1: Parent waits for all children

      Method 2: Parent checks on children in a busy-waiting loop

      Method 3: Parent waits on a semaphore as in the above code

      Justification: **(Limit: 2 Lines)**

      The parent launches one thread at a time and waits for it to complete before launching the next thread. This is equivalent to sequential processing of the divisions.

5. Modify the above code to solve the following search problem. Given an input array and a key, divide the array into K equal divisions and check if the key appears **exactly once** in each division. The program creates a thread to search each division as in the above code (gThreadCount = K). If the key appears once in **each** division, the program prints "YES"; otherwise (there is at least one division in which the key either does not appear or appears more than once), it prints "NO". Assume that the key is already stored in a global variable named gKey.
   Your program must meet the following specs:
   - There is a global array gKeyOccur of size K that holds the number of occurrences of the key in each division.
   - Each child searches its division and updates its entry in gKeyOccur. A child notifies the parent as soon as it either completes searching the entire division or determines that there is no need to search any further (the child must avoid doing unnecessary search in its division).
   - Each child must update gDoneThreadCount when it is done.
   - It is the parent's job, not the child's job to determine if the overall outcome is positive or negative and if all threads are done (unlike the above code). Each child is responsible only for the outcome of its own division.
   - The parent must print the result **as soon as it is known** and then cancel all threads and terminate cleanly. You will lose points for any delay in printing or termination.
   - The parent cannot waste CPU cycles while the child threads are searching. It must be in the waiting state until it receives a notification from a child. After processing that child's result, the parent must go into the waiting state again, waiting for the next child's notification, and so on.
   - If the hardware has x cores and x<k, only x threads can be active concurrently and the remaining k-x threads must be in the waiting state until one of the x threads is done. The point is minimizing context-switching overhead.

   Use the minimum number of shared variables and semaphores and minimize the time spent in critical sections. Give the initial value for every shared variable or semaphore that you use.
   You can reuse any lines from the given code by simply specifying their numbers; you don't have to rewrite any unchanged code.

```
sem notify=0, mutex=1, limit=X

void * Child(void* param) {
    wait (limit)

    int threadNum = ((int*)param)[0];
    int start = ((int*)param)[1];
    int end = ((int*)param)[2];

    for(int i=start; i<=end; i++) {
            if (gData[i] == gKey) {
                    gKeyOcurr[threadNum]++
                    if (gKeyOcurr[threadNum] > 1) {
                            gThreadDone[threadNum] = true;
                            post(&limit);
                            post(&notify);
                            pthread_exit(0)
                    }
            }
    }
    gThreadDone[threadNum] = true;

    sem_wait(&mutex);
    gDoneThreadCount++;
    sem_post(&mutex);
```

```
    sem_post(&limit);
    sem_post(&notify);
}
```

Parent

```
    Lines 1-9

    done = false
    while(!done) {
            wait(&notify);
            wait(&limit);
            res = CheckOccur();
            if (result == false || gDoneThreadCount == gThreadCount)
                    done = true;
            post(&limit);
    }
    Print result

    CheckOccur() {
            for (i =0; i< gThreadCnt; i++)
                    if (gKeyOccurr[i] > 1 || gThreadDone [i] == true && gKeyOccur[i] == 0)
                            return false;
            return true;
    }
```

Worksheet 5.3: Readers-Writers Problem

Consider the following solution studied in class for the **Readers-Writers Problem:**

sem_1 = 1; sem_2 = 1; counter = 0;

```
        Writer() {
          printf("\n Writer arrived");
1         wait(sem_1);
          printf("\n Performing writing");
2         perform_writing ();
3         signal(sem_1);
        }

        Reader() {
1         wait(sem_2);
          printf("\n Reader arrived");
2         counter++;
3         if (counter == 1)
                    printf("\n Checking semaphore");
4                   wait(sem_1);
5         signal(sem_2);

          printf("\n Started reading");
6         perform_ reading ();
          printf("\n Done reading");

7         wait(sem_2);
8         counter--;
9         if (counter == 0)
                    printf("\n Releasing semaphore");
10                  signal(sem_1);
11        signal(sem_2);
        }
```

1. Considering the print statements added to the above code, show the output printed by the code if the following sequence of events takes place (order is very important). Assume for simplicity that the print statements for each event execute completely before the next event occurs, that is, events are not interleaved with prints. Note that indentation in the above code is significant. So, all the indented lines after an **if** statement get executed if the condition is true.

   Reader 1 arrives
   Reader 2 arrives
   Writer arrives
   Reader 1 done reading
   Reader 3 arrives

   Reader arrived
   Checking semaphore
   Started reading

   Reader arrived
   Started reading

   Writer arrived

   Done reading

   Reader arrived
   Started reading

2. The above solution may cause starvation. Will it cause the starvation of readers or the starvation of writers? Give a specific scenario (sequence of events) that will cause starvation. **(Limit: 3 Lines)**

   It may cause the starvation of writers. A possible scenario is
   Reader arrives and locks the buffer for reading. Then writer arrives
   Then an unlimited number of readers arrive and access the buffer. Writer will starve

3. Trace the consequences of replacing signal(sem_2) on Line 11 of the **Reader** with signal(sem_1). Cover **all** the consequences and indicate whether the code will work correctly or not. If certain readers/writers get stuck, **specify the line** at which they will get stuck. You must cover two cases:

   Case 1: There is one reader in the buffer
   Impact on Readers: (**Limit: 2 Lines)**

   The reader in the buff will exit the buff but won't release the reader semaphore (sem_2). So, all subsequent readers will get blocked on Line 1 of Reader.

   Impact on Writers: (**Limit 2 Lines)**

   The reader in the buff will signal the reader/writer sem (sem_1) twice, thus allowing multiple writers to access the buff at the same time.

   Case 2: There are multiple readers in the buffer
   Impact on Readers: (**Limit: 3 Lines)**

   The first reader to complete reading will exit the buff but without releasing the reader sem (sem_2). So, all other readers in the buff will get blocked on Line 7 in the reader when they try to exit.
   Readers that arrive after the first reader exits will get stuck on Line 1 in the reader.

   Impact on Writers: (**Limit 2 Lines)**

   The first reader to exit will signal the reader/writer buff, thus making it possible for a writer to access the buffer while there are readers in the buff.

4. Modify the above code to prevent starvation with minimal changes. Show the changes on the given code.

   Sem fairness = 1;

```
   Writer() {
     wait (fairness);
1    wait(sem_1);
     signal(fairness);
2    perform_writing ();
3    signal(sem_1);
   }

   Reader() {
     wait(fairness);
1    wait(sem_2);
2    counter++;
3    if (counter == 1)
4            wait(sem_1);
5    signal(sem_2);
     signal(fairness);

6    perform_ reading ();
```

```
7         wait(sem_2);
8         counter--;
9         if (counter == 0)
10                  signal(sem_1);
11         signal(sem_2);
       }
```

Worksheet 9.2: Virtual Memory, Multiple-Choice Questions

Choose the right answer. There is only **one** correct answer.

1. Which of the following **is not necessarily** performed by the kernel in handling a page fault?
   a. Issuing a read request to the disk to fetch the missing frame into memory.
   b. Saving the state of the process that caused the page fault.
   c. Restoring the state of the process that caused the page fault.
   d. Granting the CPU to a process other than the process that caused the page fault.
   e. Updating the page table to indicate that the missing frame is now in physical memory.

2. Which of the following is (are) true about the working set size?
   a. A larger working set size increases the chances of having page faults.
   b. A larger working set size decreases the chances of having page faults.
   c. If the sum of working set sizes exceeds the number of available frames, the system will thrash.
   d. The working set size of a process remains constant throughout the process's lifetime.
   e. Both b and c are correct. f. Both a and d are correct**.** **g. Both a and c are correct.**

3. How does the page-fault frequency (PFF) technique prevent thrashing?
   a. It takes frames from a process if its page fault rate falls below a certain lower bound.
   b. It gives more frames to a process if its page fault rate falls below a certain lower bound.
   c. It takes frames from a process if its page fault rate exceeds a certain upper bound.
   d. It gives more frames to a process if its page fault rate exceeds a certain upper bound.
   e. Both a and c are true. **f. Both a and d are true.** g. Both b and c are true.

4. Which of the following is true about memory frame allocation?
   a. With global frame allocation, the execution time of a process depends on other processes.
   b. Global frame allocation does a better job at utilizing memory than local frame allocation.
   c. Local frame allocation does a better job at utilizing memory than global frame allocation.
   d. The minimum number of frames that must be allocated to a process is hardware independent.
   e. **Both a and b are true.** f. Both a and c are true. g. Both b and d are true.

5. Which of the following is **not** true about virtual memory (VM) and physical memory (PM)?
   a. VM allows the OS to load more processes in memory, thus giving more options to the scheduler.
   b. VM decreases the amount of I/O needed.
   c. Implementing VM does not require any hardware support.
   d. With VM, a program can be run even if the size of its logical address space exceeds the PM size.
   e. A VM system may load into physical memory an *instruction* that the program will never execute.
   f. A VM system may load into physical memory a *data element* that the program will never access.

Worksheet 9.1: Virtual Memory, Problem Solving

Q1. Consider a virtual memory system with 8 pages (0 through 7) and 3 frames (0, 1, 2). Trace the state of the system for the sequence of page accesses shown below using the Least Recently Used (LRU) page replacement algorithm. Show the contents of the three frames after each page request (as done in class), and then give the number of page faults.

	6	2	6	5	5	4	1	2	3
f1	6	6	6	6	6	6	1	1	1
f2		2	2	2	2	4	4	4	3
f3				5	5	5	5	2	2

Number of page faults: 7

Q2. Consider a demand-paging system with a paging disk that has an average access and transfer time of 8 milliseconds. Addresses are translated through a page table in **main memory**. Main memory access time is 5 microseconds per access. The system also has a Table-Lookaside Buffer (TLB) with a negligibly small access time. Assuming that 90% of the memory accesses hit in the TLB, all the accesses that hit in the TLB do not cause page faults and only 2% of the accesses that miss in the TLB (0.2% of the total) result in page faults, compute the **effective access time (EAT)** for a memory operation. Make sure that you account for the right number of memory accesses in the case of a TLB miss. You may neglect the memory access time when a page fault happens. First, calculate the time for each of the three different cases shown below, and then use those times to calculate the EAT. Show your calculations **clearly**. A final answer with no calculations will get zero credit even if it happens to be correct.

Time for accesses that hit in the TLB = 0.9 x M = 0.9 x 5 = 4.5 μs

Time for accesses that miss in the TLB and don't cause page faults = 0.1 x 0.98 x 2M
= 0.098 x 2 x 5 ≈ 1 μs

Time for accesses that miss in the TLB and cause page faults = 0.1 x 0.02 x 8 x 10^3 = 16 μs

Effective access time = 4.5 + 1 + 16 = 21.5 μs

Q3. A program has one loop. The code outside the loop accesses 200 pages and the code inside the loop sequentially accesses 10 pages. The loop is executed 50 times, and all 10 pages are accessed sequentially in every iteration of the loop. Assume for simplicity that the pages accessed inside the loop are distinct from the pages accessed outside the loop (no overlap). The program accesses 100 words in each page. Assuming **pure demand paging** and the **LRU** page replacement policy, calculate the page fault rate for each of the following cases. Show your calculations **clearly**.

1. The process has 12 frames allocated to it.

$$\frac{Number\ of\ page\ faults}{Number\ of\ memory\ accesses} = \frac{200+10}{100(200+50\times 10)} = \frac{210}{70000} = \frac{3}{1000}$$

2. The process has 8 frames allocated to it.

$$\frac{Number\ of\ page\ faults}{Number\ of\ memory\ accesses} = \frac{200+10\times 50}{100(200+50\times 10)} = \frac{700}{70000} = \frac{1}{100}$$

Worksheet 4.1: Multiple-Choice Questions

In the questions below, circle the right answer. There is only one correct answer

1. Which of the following is (are) true about threads?
   a. Threads within the same process must be run on the same CPU.
   b. Threads within the same process share global variables.
   c. Threads within the same process share the same stack.
   d. Parallel programming using threads can utilize more cores than parallel prog. using processes.
   e. Both b and c are correct. f. Both b and d are correct. g. Both a and c are correct

2. Which of the following is true about threads and processes?
   a. Context switching between threads is faster than context switching between processes.
   b. Threads within the same process share global variables, but different processes can't share global variables.
   c. A thread generally uses more resources than a process does.
   d. A multi-core system can be utilized using multiple threads but can't be utilized using multiple processes.
   e. **Both a and b are correct.** f. Both a and d are correct g. a, b and d are correct.

3. What is (are) the advantage(s) of dividing an application into multiple threads relative to dividing it into multiple processes?
   a. Utilizing a multi-core system b. One slow task won't slow the whole application
   c. Using less resources d. Easier communication using global variables
   e. Both a and c are correct **f. Both c and d are correct** g. Both b and d are correct

4. What are the limitations of Amdahl's Law?
   a. It assumes that the parallelizable code can be divided *equally* among the CPUs.
   b. You cannot apply it to a system with more than 10 CPUs.
   c. It does not account for the communication or synchronization overhead.
   d. You can apply it only when all CPUs are on the same chip not on different chips.
   e. Both a and b are correct. **f. Both a and c are correct.** g. Both a and d are correct.

5. In a given program, one fifth of the code is parallelizable. What's the maximum speedup factor that can be achieved on a quad-core processor under ideal conditions?
   a. 4 **b. 20/17** c. 20 d. 2.5 e. 5 f. 5/4 g. ¼

   Speedup = 1 / ((1/5)/4 + 4/5) = 1 / (1/20 + 16/20) = 20/17

6. Which of the following is (are) true about concurrency and parallelism?
   a. With concurrency, only one process may be *running* at a given point in time, while with parallelism multiple processes may be *running* simultaneously.
   b. With concurrency, the total time needed to execute a given set of long CPU bursts from different processes is always less than the time needed to execute the processes sequentially.
   c. Parallelism is achieved using multithreading, while concurrency is achieved using multiprocessing.
   d. Concurrency requires multiple CPUs, while parallelism may be achieved on a single CPU.
   e. Both a and d are correct. f. Both a and c are correct. g. a, c and d are correct.

Worksheet 5.2: Multiple-Choice Questions

In the questions below, circle the right answer. There is only one correct answer

1. What's the critical-section problem?
   a. It's a synchronization problem that is unique to the bounded-buffer problem.
   b. It's an open problem in Computer Science, and there are currently no known solutions to it.
   c. It's a synchronization problem that occurs when multiple parallel processes try to update a shared variable at the same time.
   d. It's a problem that can be solved only using semaphores.
   e. Both b and c are correct. f. Both a and c are correct. g. Both c and d are correct.

2. Which of the following is (are) true about locks and semaphores?
   a. A semaphore is more powerful than a lock.
   b. A lock is more powerful than a semaphore.
   c. A lock is binary, while a semaphore can have more than two values.
   d. Semaphores are used only on single-core systems, while locks are used only on multi-core systems.
   e. Both b and c are correct. **f. Both a and c are correct.** g. Both a and d are correct.

3. Which of the following is (are) true about spin locks and mutex locks?
   a. A process waiting on a spin lock uses CPU cycles but a process waiting on a mutex lock does not.
   b. Using mutex locks involves more context switching.
   c. Using spin locks is more efficient on a single-CPU system.
   d. Using spin locks is more efficient when the process waits for a long time.
   e. Both b and c are correct. f. Both a and d are correct. **g. Both a and b are correct**.

4. How does protecting a critical section (CS) with a semaphore ensure mutual exclusion?
   a. When a process is in its CS, it never loses the CPU until it completes the CS.
   b. When a process is in its CS, all other processes are placed in the waiting state.
   c. When a process is in its CS, no other process is allowed to be in a CS.
   d. When a process is in its CS, any process that tries to access a CS that is protected by the same semaphore is placed in the waiting state.
   e. Both b and d are correct. f. Both c and d are correct. g. Both b and c are correct.