Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

1. Consider the following C program, with line numbers:

```
1     #include <stdio.h>
2
3     int main() {
4       fork();
5       fork();
6       fork();
7     }
```

How many processes are created as the result of any `fork()` within the above code?

Each fork doubles the amount of processes there are by creating a child process for each existing process.. There are 7 processes created for a total of 8 processes. The fork on line 4 makes one process, the fork on line 5 makes two processes, and the fork on line 6 makes four processes.

2. Consider the following C program, with line numbers:

```
1     #include <stdio.h>
2
3     int main() {
4       if (fork())
5         if (fork())
6           fork();
7     }
```

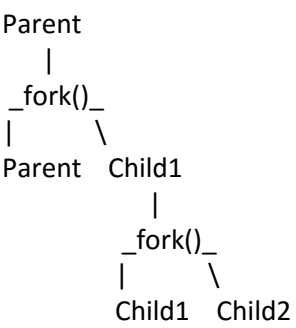
How many processes are created as the result of any `fork()` within the above code?

7 processes are created as a result of all the forks for a total of 8. The fork in line 4 creates one processes for a total of two, then the fork in line 5 creates one processes as well for a total of four. Then the last fork on line 6 creates 4 processes for a total of 8.

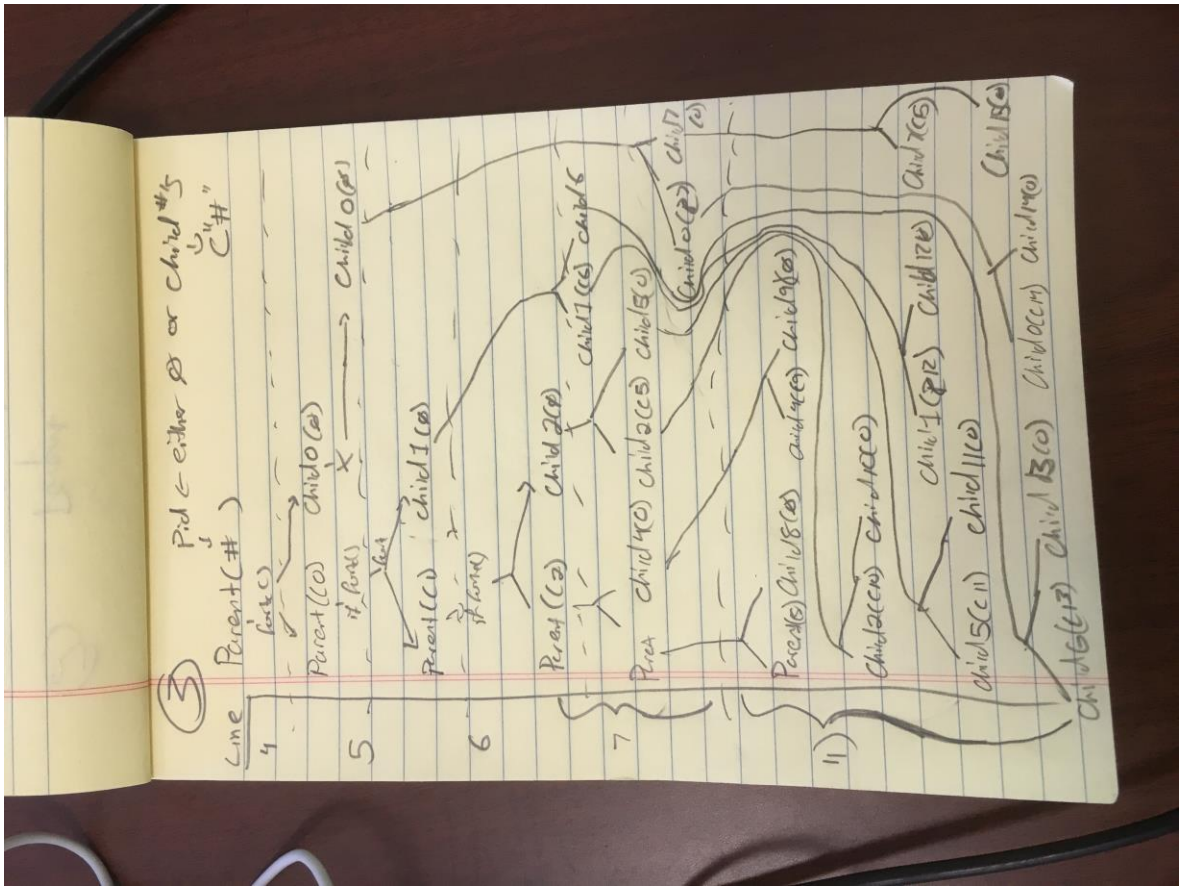
3. Consider the following C program, with line numbers:

```
1     #include <stdio.h>
2
3     int main() {
4       if (fork())
5         if (fork())
6           if (fork())
7             fork();
8           else
9           ;
10        else
11          fork();
12
13    }
```

Random example tree – Not the answer



Please draw a tree depicting the parent-child relationship in the code above at some point in time where the original process and all `fork()`ed processes exist.



Name: Pid: @ucsd.edu email:

4. [CMU 15-213] Consider the following C program, with line numbers:

```
1     int main() {
2       int counter = 0;
3       int pid;
4
```

```
5       if !(pid = fork()) ) {
6         while((counter < 2) && (pid = fork()) ) {
7           counter++;
8           printf("%d", counter)
9         }
10        if (counter > 0) {
11          printf("%d", counter);
12        }
13      }
14
15      if(pid) {
16        waitpid(pid, NULL, 0);
17        counter = counter << 1;
18        printf("%d", counter)
19      }
20    }
```

Use the following assumptions to answer the questions:

- All processes run to completion and no system calls will fail.
- printf() is atomic and calls fflush(stdout) after printing argument(s) but before returning.
- Logical operators such as && evaluate their operands from left to right and only evaluate the smallest number of operands necessary to determine the result.

A. List all possible outputs of the program in the following blanks. (You might not use all the blanks.)

1	0	1
2	1	
4	2	

B. If we modified line 10 of the code to change the > comparison to >=, it would cause the program flow to print out zero counter values. With this change, how many possible outputs are there? (Just give a number, you do not need to list them all.)

NEW NUMBER OF POSSIBLE OUTPUTS = 2

Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

5. [Voelker] The Intel x86 instruction set architecture provides an atomic instruction called XCHG for implementing synchronization primitives. (If you are curious, this reference page shows the full syntax and semantics of the instruction.) Semantically, XCHG works as follows (although keep in mind it is executed atomically):

```
1     void XCHG (bool *x, bool *y) {
2       bool *tmp = *x;
3       *x = *y;
4       *y = tmp;
5     }
```

Given this instruction, please define, in c-like pseudo-code, the following operations:

- `mutex_acquire(bool mutex)  # Acquire lock`
  - `private bool *locked = true;`
  - `#locked prevents other threads from running the included code`
- `mutex_release(bool mutex)  # Release lock`
  - `private bool *locked = false;`
  - `#unlocked, allowing other threads access`

6. Please consider the following code:

```
1     volatile int counter = 0; /* global */
2
3     int main() {
4       int iters = 1000;
5
6       pthread_t t1, t2;
7
8       pthread_create(&t1, NULL, thread, &iters);
9       pthread_create(&t2, NULL, thread, &iters);
10
11      pthread_join(t1, NULL);
12      pthread_join(t2, NULL);
13
14      /* Did a race condition reveal itself? */
15      if (counter != (2*iters)) printf("RACE CONDITION OBSERVED\n”);
16      else printf("NO RACE CONDITION SEEN\n”);
17    }
18
19    void *thread(void *arg) {
20      int i,
21    ` mutex_acquire(bool mutex);
22      int iters = *((int *)arg);
23
24      for (i = 0; i < iters; i++)
```

```
25          cnt++;
26        mutex_release(bool mutex);
27        return NULL;
28      }
```

The code above has a concurrency control problem.

A. What is the critical resource?
    a. A resource that can only be used by one process at a time. Int iters is the critical resource.

B. What is the critical section?
    a. The portion of a multi-process program that cannot be executed by more than one process. The void *thread method is the critical section.

C. Recall the concurrency control primitives you defined in (5) above. Please modify the above code as needed to use them to protect the critical section, correcting the above code.

Name: Amit Nijjar Pid: A11489111 email: a2nijjar@ucsd.edu

CSE 120 Homework #2
Spring 2016 (Kesden)

7. Let's consider the busiest aisle of the grocery store, where the extra-savings packages of Ramen noodles are stored. The area from which the noodles are accessible can accommodate **at most one person with a shopping cart, or at most two people with hand-baskets, but not both**. When the prescribed maximum number of people are in the Ramen area, those waiting to select their favorite flavors of Ramen noodles must wait in other places along the aisle. As those making their selections move on, the nearby waiting shoppers then take their places in the Ramen area. The waiting policy is in informal. **It shouldn't be intentionally unfair – but no one is taking numbers** in the Ramen section, either. Instead, it should ensure that there is **no deadlock**, and **that someone steps up to the Ramen aisle, whenever possible**.

   Please implement a **mutex-based** solution to the concurrency control problem described above. In other words, implement the following entry functions, as well as any necessary global initialization function to ensure that the shoppers get access to the Ramen section as prescribed, **using only simple mutexes for concurrency control** [mutex_var, mutex_intit(…), mutex_acquire(…), mutex_release(…)] **.**

   Your solution should be written in pseudo-code similar to an object-oriented or imperative high-level language.

   Your solution should **contain exactly the following** functions to control access to the Ramen noodles**.** In addition, you may have as many helper functions as you choose:

   - `GlobalInitialization()  /* This runs before any thread and all variables initialized within are global */`
   - `LargeCartEnter()`
   - `LargeCartExit()`
   - `SmallBasketEnter()`
   - `SmallBasketExit()`

```
class shoppingCart
{
        public static void main (String[] args){
        }

        public globalInitialization(){
                boolean waitingLine;
                int inIsle;
                while(waitingLine == true){
                        if(inIsle == 0){
```

```
                        largeCartEnter();
                        largeCartExit();
                        smallBasketEnter();
                        smallBasketEnter();
                        smallBasketExit();
                        smallBasketExit();
                    }
                    else{
                        smallBasketEnter();
                        smallBasketExit();
                    }
                }
            }
            Public largeCartEnter(){
                enter();
                shop();
            }
            Public largeCartExit(){
                exit();
            }
            Public smallBasetEnter(){
                enter();
                shop();
            }
            Public smallBasketExit(){
                exit();
            }
}
```

Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

8. Please consider the pseudo-code on the next couple of pages. Assume that each of the function shown is executed independently by one or more threads within the same task. As written, the code below is unsafe - critical regions are left unprotected. It is also broken in that the circular buffers can over-flow and under-flow.

   Please modify the code by adding semaphores and semaphore operations, as necessary to resolve any concurrency concerns, e.g. races, etc. Please also correct the buffer logic (and any other errors you might find).

   Please use semaphores with exactly the following operations:

   - Semaphore (int count) // initializes a new semaphore with a count of c
   - P() // the traditional "wait" operation
   - V() // The traditional "increment" operation

For example:

```
Semaphore newSem = new Semaphore (2);
newSem.P();
newSem.V();
```

Please also modify the code to correct any other problems that might exit, for example in the implementation of the shared buffer logic.

SEE NEXT PAGE FOR THE CODE ASSOCIATED WITH THIS QUESTION. THANKS!

Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

Below is the code for question #8 on the prior page:

```
const int BSIZE = 1024;

BreadSlice breadSupply[BSIZE];
Meat meatSupply[BSIZE];
Sandwhich sandwichSupply[BSIZE];

int bread_index = 0;
int meat_index = 0;
int sandwich_index = 0;


void BreadBakery()
{
  Semaphore breadSem = new Semaphore(0);
  while (FOREVER)
  {
   breadSupply[bread_index++] = new BreadSlice(); /*Acquire more bread */
   breadSem.V();
  }
}

void MeatFactory()
{
  Semaphore meatSem = new Semaphore(0);
  while (FOREVER)
  {
   meatSupply[meat_index++] = new MeatSlice(); /* Acquire more meat */
   meatSem.V();
  }
}


void SandwichFactory()
{
  // Please assume that someone, somewhere else is eating sandwiches by
  // removing it from the sandwichSupply in a way that is consistent with
  // sharing discipline that you define. In other words, make appropriate
  // adjustments here and assume that they are made in the code you cannot
  // see, as well.

  while (FOREVER)
  {
    Semaphore foodSem = new Semaphore(0);
    sandwichSupply[sandwich_index++] = /* Build a sandwich including… */
      new Sandwich(breadSupply[--bread_index]  /* Top slice of bread */
                   breadSem.P();
                   meatSupply[--meat_index],   /* Piece of meat */
                   meatSem.P();
```

```
                    breadSupply[--bread_index]);/* Bottom slice of bread */
                    breadSem.P();
                    foodSem.V();
    }
  }

Void eat(){
       While(true){
              If(//sandwhich is eaten){
                    foodSem.V();
              }
       }
}
```

Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

9. Consider the problem of sharing a partitioned multi-purpose room among groups. The room is very flexible in that it is divided into three separate sections, each of which can be scheduled independently. This allows the room to support four different types of events: plays, dances, speeches, and dinners.

```
enum section = {SECTN_0, SECTN_1, SECTN_2};
enum event = { PLAY, DANCE, SPEECH, DINNER };
typedef struct
{
  bool section0;
  bool section1;
  bool section2;
} mproom; /* Set of sections */
```

For a visual, the multi-purpose room is layed out, as below:

stage	section 0	section 1	section 2

Depending on the event all or only a piece of the room may be required. At most one event may take place in any given section of the room at a time. Furthermore, each event has different requirements:

- PLAYs are well-attended and require the use of all 3 sections.
- DANCEs require two contiguous sections.
- SPEECHes require section 0 because they make use of the stage.
- DINNERs require any one section.

Using pseudo-code please design a monitor that reserves the necessary sections of the room for events. Your solution should maximize the usage of the monitor as much as possible without violating any of the constraints above. **Be sure to specify which of the monitor semantics (Brinch Hanson, Mesa, or Hoare) are implemented by your monitor.**

Your monitor should provide exactly the following entry functions:

- Entry mproom event_begin (event e) /* Given an event, returns set of sections assigned for event */
- Entry void event_end (event e, mproom alloced_sections) /* Undoes above*/

The event_begin() function reserves and returns a set of sections that satisfies the requirements for the event. If there is no such set available, event_begin() blocks until one is available.

To specify this set, event_begin() will return an mproom, which is a structure containing one boolean flag for each section. A value of true for a section indicates that the section has been reserved to satisfy the request, false indicates that the section is in use by, or available for, another request. You may assume that the array is copied upon return from event_begin(), so you may return a locally declared array, if you choose.

The event_end() function releases the alloced_sections that were used by event. You may assume that event_end() is always used correctly -- that is, a thread will only execute event_end() after a previous call to event_begin() that returned alloced_sections.

Hoare monitor scheduler {

```
enum section = {SECTN_0, SECTN_1, SECTN_2};
enum event = { PLAY, DANCE, SPEECH, DINNER };
typedef struct
{
  bool section0;
  bool section1;
  bool section2;
} mproom; /* Set of sections */
```

```
While(there are events){
      If(event == play){
            If(section0 == 0 && section1 == 0 && section2 == 0){
                  event_begin();
                        becomeCivilized();
                  event_end();
            }
            else{
                  event.signal(play);
            }
      }
If(event == dance){
            If(//any two next to each other are empty){
                  event_begin();
                        party();
                  event_end();
            }
            else{
                  event.signal(dance);
            }
      }
```

```
If(event == Speech){
            If(section0 == 0){
                  event_begin();
                        listen();
                  event_end();
            }
            else{
                  event.signal(speech);
            }
      }

If(event == dinner){
            If(section0 == 0 || section1 == 0 || section2 == 0){
                  event_begin();
                        becomeFat();
                  event_end();
            }
            else{
                  event.signal(dinner);
            }
      }

}



}
```

Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

10. Given the following execution profiles, please fill in the CPU occupancy chart on the next page and compute the requested metrics for each scheduling discipline. Simply shade or color a box if the resource it represents is occupied during the time slice it represents.

Process X	**Process Y**
CPU-25ms	CPU-15ms
Disk-10ms	Network-10ms
CPU-25ms	CPU-15ms
Network-5ms	Network-5ms
CPU-30ms	CPU-15ms
Disk-10ms	Disk-10ms
CPU-15ms	CPU-20ms
Disk-10ms	Network-5ms
CPU-5ms	CPU-25ms
Network-5ms	Network-10ms
CPU-25ms	CPU-20ms
Network-15ms	Disk-10ms
	CPU-20ms

Assumptions

- Please assume that *Process Y* is admitted 1mS after *Process X* is admitted.
- Please also assume that whereas the CPU is preemptible, the disk and network are not -- each burst of I/O should occur contiguously.
- Furthermore, please assume that each I/O burst is a blocking operation and is requested by the last instruction or instructions of the prior CPU burst.

Metrics

For each scheduling discipline, please compute each of the following:

- The percent utilization of each resource (CPU, disk, network) over the aggregate lifetime of both processes.
- The turn-around time for each process.
- The amount of time that each process spends runnable, but not running.

Scheduling Disciplines

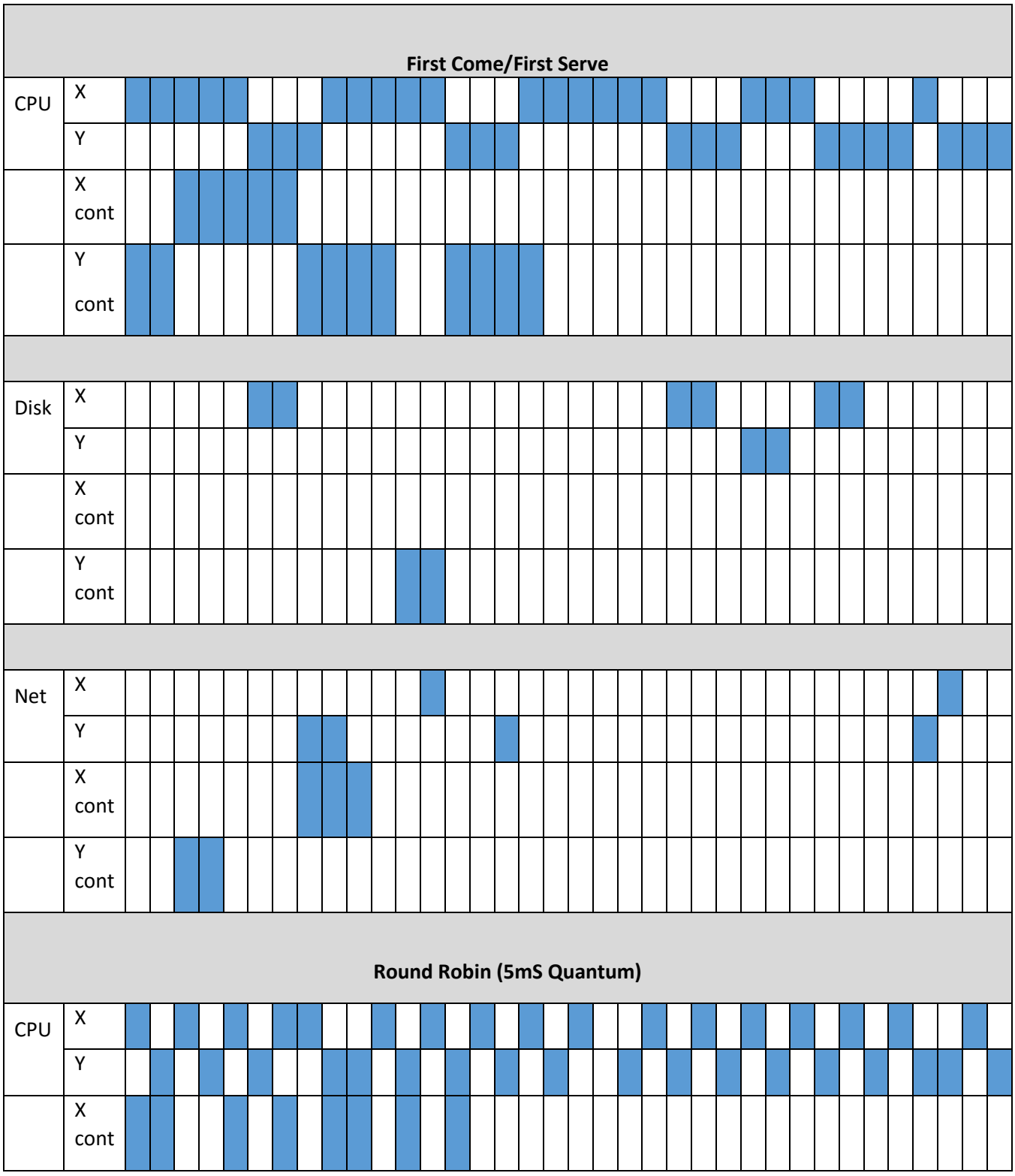
- First Come-First Serve (FCFS) (no preemption, but can context-switch upon I/O blocking)
- Round-robin (5ms quantum)
- Round-robin (10ms quantum)
- Round-robin (20ms quantum)

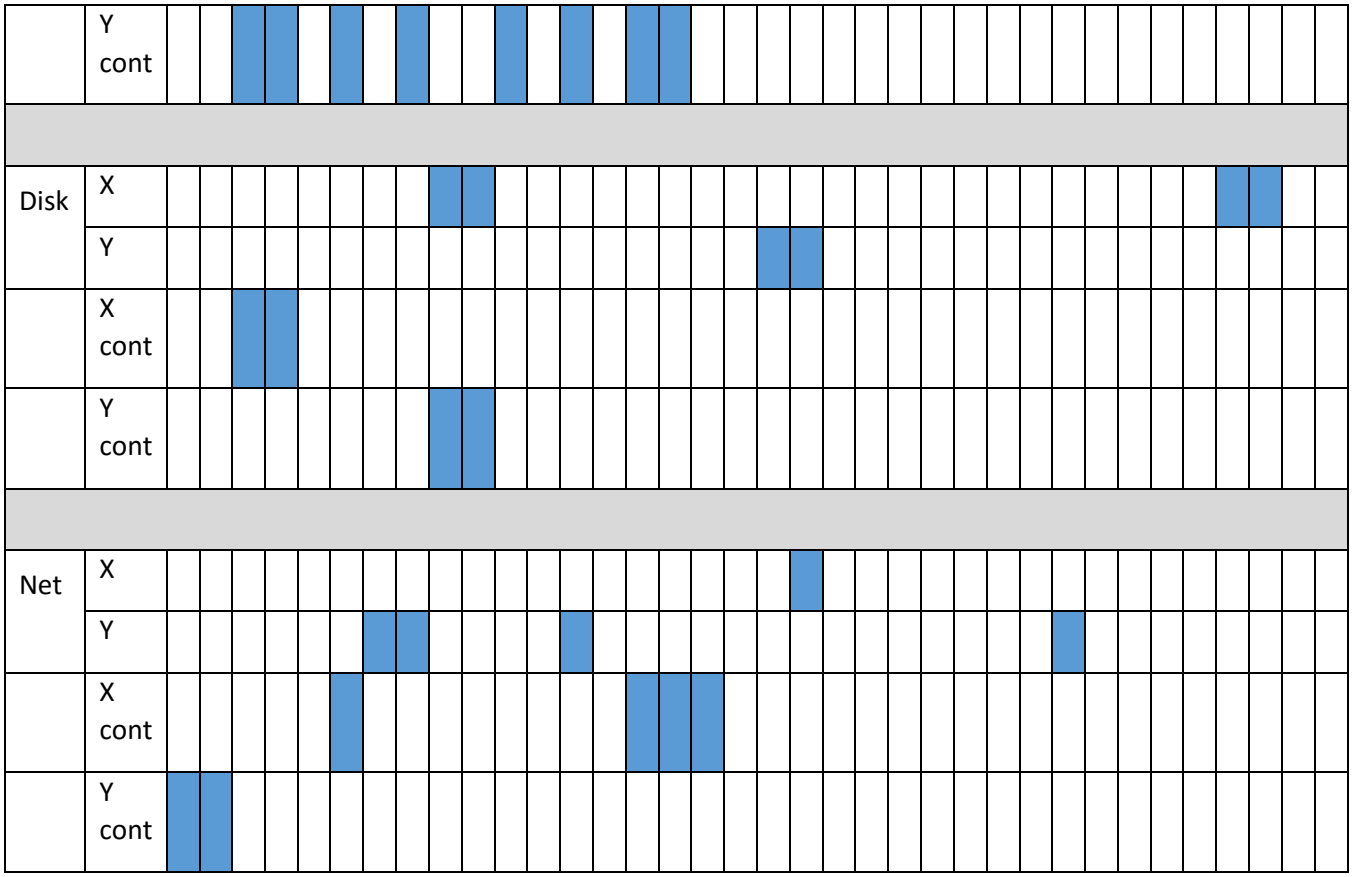
PLEASE SEE NEXT TWO PAGES FOR CHARTS. THANKS!

Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

Chart #1 of 2 for Question 10



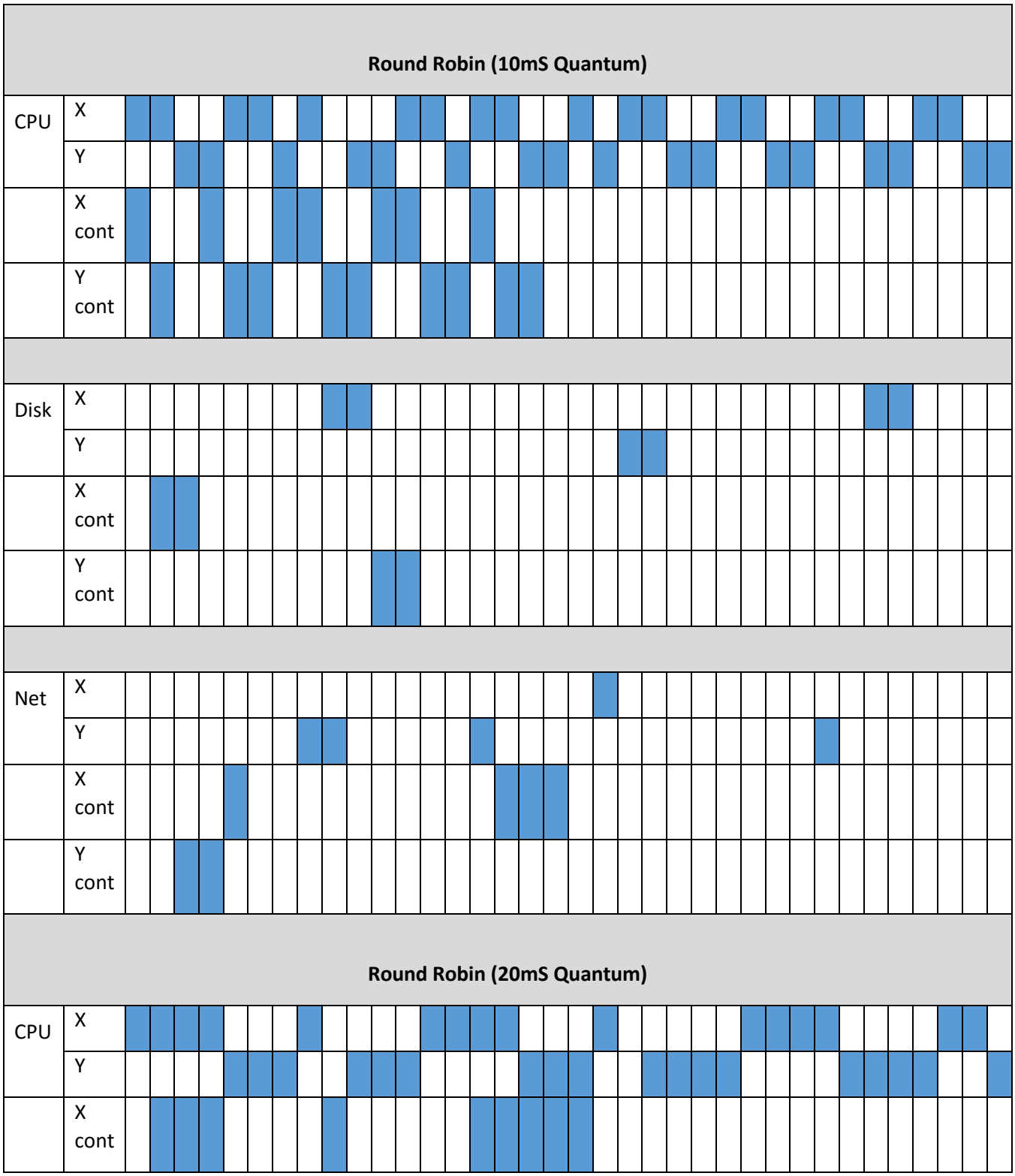

Y cont
Disk
X
Y
X cont
Y cont
Net
X
Y
X cont
Y cont

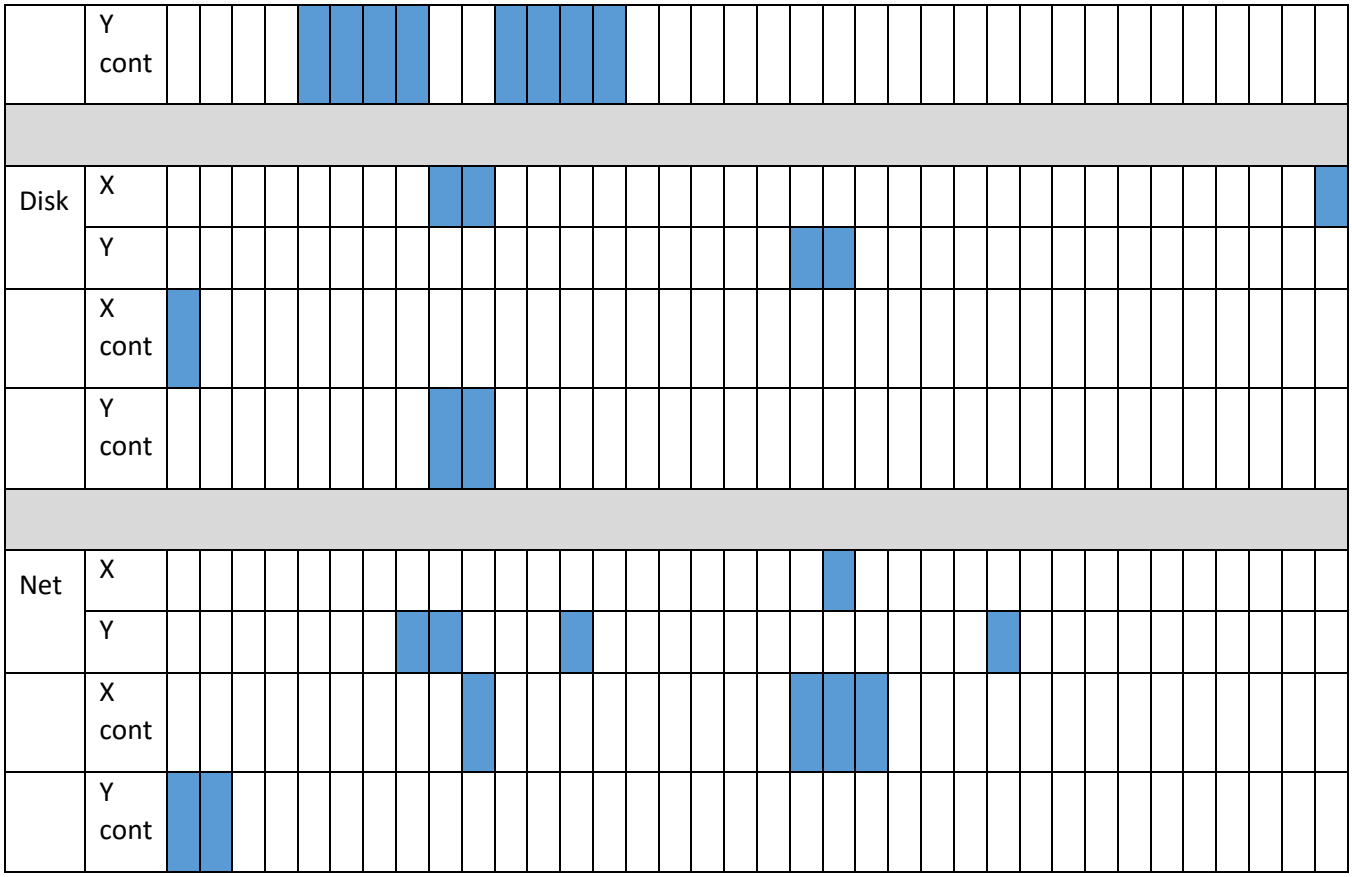
Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

:

Chart #2 of 2 for Question 10





1. FCFS:
    a. CPU = 51/53 = 96.2%
    b. Disk = 10/53 = 18.9%
    c. Net = 11/53 = 20.8%
2. Round-Robin (5ms):
    a. CPU = 51/53 = 96.2%
    b. Disk = 10/53 = 18.9%
    c. Net = 11/53 = 20.7%
3. Round-Robin (10ms):
    a. CPU = 51/54 = 94.4%
    b. Disk = 10/54 = 18.5%
    c. Net = 11/54 = 20.4%
4. Round-Robin (20ms):
    a. CPU = 51/58 = 87.9%
    b. Disk = 10/58 = 17.2%
    c. Net = 11/58 = 18.97%

5. FCFS:
    a. X = 46 * 5 = 230ms
    b. Y = 53 * 5 = 265ms

6. Round-Robin (5ms):
    a. X = 53 * 5 = 265ms
    b. Y = 52 * 5 = 260ms
7. Round-Robin (10ms):
    a. X = 54 * 5 = 270ms
    b. Y = 53 * 5 = 265ms
8. Round-Robin (20ms):
    a. X = 58 * 5 = 290ms
    b. Y = 50 * 5 = 250ms

9. FCFS:
    a. X = 10 * 5 = 50ms
    b. Y = 17 * 5 -1 = 84ms
10. Round-Robin (5ms):
    a. X = 17 * 5 = 85ms
    b. Y = 16 * 5 - 1 = 79ms
11. Round-Robin (10ms):
    a. X = 18 * 5 = 90ms
    b. Y = 17 * 5 - 1= 84ms
12. Round-Robin (20ms):
    a. X = 22 * 5 = 110ms
    b. Y = 14 * 5 - 1 = 79ms

Name: Amit Nijjar Pid: A11489111 a2nijjar@ucsd.edu email:

CSE 120 Homework #2
Spring 2016 (Kesden)

13. [Silberschatz via Voelker] Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. What is the CPU utilization for a round-robin scheduler when:

    A. The time quantum is 1 millisecond
        - (Context switching overhead / (context switching overhead + quantum time)) x 100 = (.1/1.1)x100 = 91%
    B. The time quantum is 10 milliseconds
        - 10x1.1 + 10.1 / 21.1 = 94%

14. [Voelker] Annabelle, Bertrand, Chloe and Dag are working on their term papers in CSE 120, which is a 10,000 word essay on *My All-Time Favorite Race Conditions*. To help them work on their papers, they have one dictionary, two copies of Roget's Thesaurus, and two coffee cups.

- Annabelle needs to use the dictionary and a thesaurus to write her paper;
- Bertrand needs a thesaurus and a coffee cup to write his paper;
- Chloe needs a dictionary and a thesaurus to write her paper;
- Dag needs two coffee cups to write his paper (he likes to have a cup of regular and a cup of decaf at the same time to keep himself in balance).

Consider the following state:

- Annabelle has a thesaurus and needs the dictionary.
- Bertrand has a thesaurus and a coffee cup.
- Chloe has the dictionary and needs a thesaurus.
- Dag has a coffee cup and needs another coffee cup.

    A. Is the system deadlocked in this state? Explain using a resource allocation graph as a reference.
        - The system is not in a deadlocked state, although most people are waiting for a resource to free up, Bertrand has everything he needs. The others must wait for him to finish and once he is done, resources will be freed and Chloe and Dag will have a chance to write their papers.

People\resource	dictionary	Thesaurus	Thesaurus	Coffee cup	Coffee cup
Annabelle	needs	Needs /has			
Bertrand			Needs /has	Needs /has	
Chloe	Needs /has		needs		
Dag				needs	Needs /has

        -
    B. Is this state reachable if the four people allocated and released their resources using the Banker's algorithm? Explain.

- Yes. This state is possible using the banker's algorithm. The banker's algorithm allows threads to take resources when those resources become available. It does not allow threads to take resources that do not exist. The algorithm allows resources to be taken as needed, not distributed evenly. The Bertrand thread will terminate upon completion, from there, the resources will be distributed to the other members of the group until they are able to finish their assignment. Threads take resources and keep on taking them until they have enough to do what they need to do using the banker's algorithm.