1. Assume processes P_1...P_n, n>=2, if the receive primitive specifies P_1 as the process to receive from and it stops sending message, the all other communication is blocked and every other process' (P_n, n !=1) message is kept on hold until P1 sends another message before the receive channel opens up.
   The system could end up deadlocked because all process are now waiting on a single process in order to resume execution and the whole system could stall otherwise.

2. Total => A = 3; B = 14; C = 12; D = 12;
   a.

	Allocation	**Max**	**Available**	**Need**
	ABCD	ABCD	ABCD	ABCD
P0	0012	0012	1520	0000
P1	1000	1750		0750
P2	1354	2356		1002
P3	0632	0652		0020
P4	0014	0656		0642

   b. No, the state can be considered unsafe. Although the next series of request and allocation may not lead to deadlock, the total resources needed is much less than the resources available and depending on the order, this *may* lead to a deadlock.

   c. Yes. The requested resources are available for use and can be allocated to P1

3. **Deadlock**: All processes/threads waiting on a signal from other processes, usually resources have been released for use.
   Livelock: In this case, all threads/processes are in waiting (resource request) state however this is not because the resources are being used but rather the processes are alternating between active and waiting because they're waiting for another process to proceed
   Starvation: This is the case where a processes or number of processes are never allowed access to the resources in order to execute because they are constantly bypassed by newer processes.

4.
   a. One instance where these conditions cannot cause deadlock will be a case when the total needed resources for all processes/threads involved is less than what's available

b. When the order of resource request and assignment is such that each process has requested and

5.

a. Yes, the situation can be considered a deadlock
b. A communication deadlock.

6.

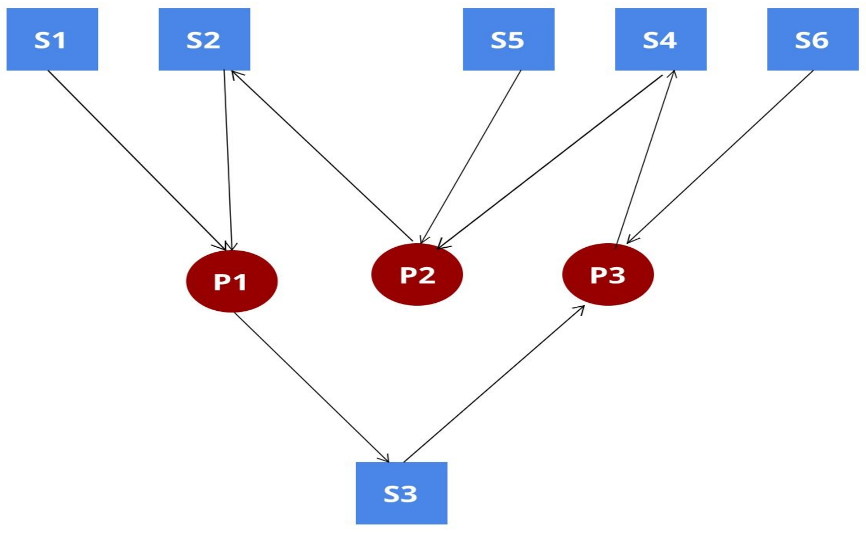
a. In the worst case scenario, i = **max** and o = 0; **O** is left waiting for **P** to process some of **i** in order to generate **o** for consumption, this is not possible since **I** continually adds to the disk with no clear limit set on how much of the disk, **max**, it can use.
   P can no longer add to the disk and **O** is waiting to consume the output that never comes..
   I is waiting on **P** to consume **i** so it can add to the disk, **P** is waiting on **O** to consume to created disk space and **O** is waiting on **P** to add to **o** so it can consume.
   The deadlock is essentially caused by lack of proper control over disk space.

b. Obviously mutual exclusion has to be implemented such that only one process can make changes to the disk at any one time.
   Instead of allowing **I** to generate to **i** as long as there's diskspace, and P to generate **o** as long as there's diskspace, a method in the critical section should always limit the maximum amount of disk space each buffer is allowed to occupy.
   So, at a certain point, **I** should pause so that **i** can either be emptied out or slowly decreased to a safe level and then normal operations can be resumed.

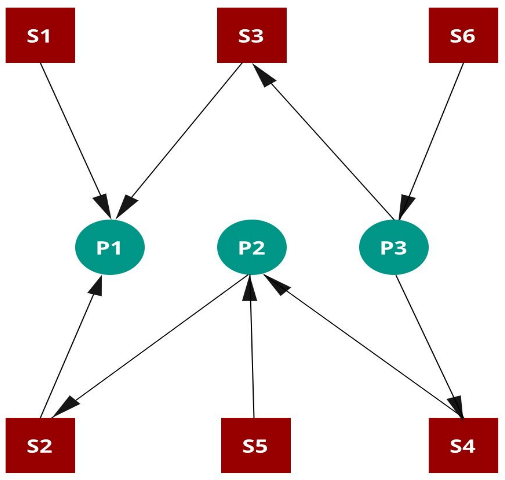
7.

a. In the event that the resource are requested and assigned in the exact order but with each process taking a turn to request, this would be the outcome.
   There's **circular wait caused with P1 waiting on P3 for S3, which is in turn waiting for P2 for S4 which is waiting on P1 for S2**. The deadlock

never breaks.



b. The simplest solution would be to reorder the order of requests for one of the processes such that it requests resources it has in common with other processes last. In this instance, have P3 request S4 and S3 after it has requested and received S6. The order then becomes get(s6), get(s4), get(s3) This way, holding the resource still prevents a deadlock with the worst case scenario shown in the diagram below:



In the worst case, we are guaranteed the processes will execute in the order or P1, P2 then P3 to break the deadlock.

Self Evaluation

1. Yes. In order to avoid race conditions and also stop threads from overtaking other threads already waiting in the queue, the signal method is used to notify threads in waiting. Also, threads have a limited time they're allowed to hold onto the forks so there shouldn't be any case of "indefinite eating" or circular wait on the forks. Finally, the reentrant lock is wrapped in a try-catch-finally block to ensure that in the cause of interruption, the critical section is still released for the other philosophers to proceed.

2. The assignment was completed and I'm expecting upward 90% of the grade. I stayed in class and got the solutions for most problems as they were discussed. For the others, I believe my answers were fairly reasonable.
3.
    a. Assessing the question difficulty based on previous assignments. I wasn't able to implement the threading the the last on successfully so I did some extra reading and also decided to employ the Soonest Job First approach to the assignment.
    b. Visualizing some of the systems described took time and some of the topics had not yet been discussed but I followed as they were discussed in class. This helped to break down the questions.
4.

Date	Activities	Hours	Outcome
Oct 29	2 hours spent reading up on the material and resources online. Java 8 documentation on the reentrant lock and conditions as well as sample codes. Read over the dining philosophers' problem and made diagrams to mimic the class the server object.	2	Had a general idea on how to do the coding problem and also got a better understanding of threads.
Nov 1-6	Worked on questions during lectures as the topics were discussed and wrote down the answers immediately after the lecture. Managed to finish most essay type questions beside 6. Started work on pseudocode.	Varied	
Nov 6	Started writing code.	3	Created classes. Implemented the required interfaces. No testing done but there's a problem getting the threads to run and implement the lock properly
Nov 7	Continue working on the program. Find a solution to the non-working threads and also come up with something to stop starvation of philosophers. Test the solution using various scenarios described in the readme.txt file and generate the sample output.	4	Problems 1, 2b, 6 & 3 a remain unsolved. But the codes working fine although I'm not sure how to slow down the

			output. None of the philosophers are starving and I even found a way to make the output varied.
Nov 10	Finished the last essay type questions.	4	Finished and submitted assignment.