COMP9334 Project, Term 1, 2023: Priority queueing for multi-phase jobs

Due Date: 5:00pm Friday 21 April 2023

Version 1.02

Updates to the project, including any corrections and clarifications, will be posted on the course website. Make sure that you check the course website regularly for updates.

Change log

- Version 1.02 (11 April 2023) The changes are in magenta color and can be found on Pages 9, 21 and 23.
- Version 1.01 (25 March 2023) The changes are:
  - There were errors in Section 4.2 and this section has been reduced to remove the errors.
  - Sections 4.3 and 4.4 have been added.
  - Some parameters for the design problem in Section 5.2 have changed, see the text typeset in blue.
  - There are also changes in Section 6.1.4 and 6.4, see the text in blue and the yellow highlight.
  - You should download `sample_project_files_25Mar.zip` and **discard** the earlier version.
- Version 1.00. Issued on 20 March 2023.

1 Introduction and learning objectives

When you were learning about operational analysis earlier in the term, we talked about jobs that require multiple visits to the CPU (or servers) to receive their service. In this project, you will use simulation to study how priority queueing can be used to improve the performance of a multi-server system that works on jobs that require multiple visits to the servers.

In this project, you will learn:

1. To use discrete event simulation to simulate a computer system
2. To use simulation to solve a design problem
3. To use statistically sound methods to analyse simulation outputs

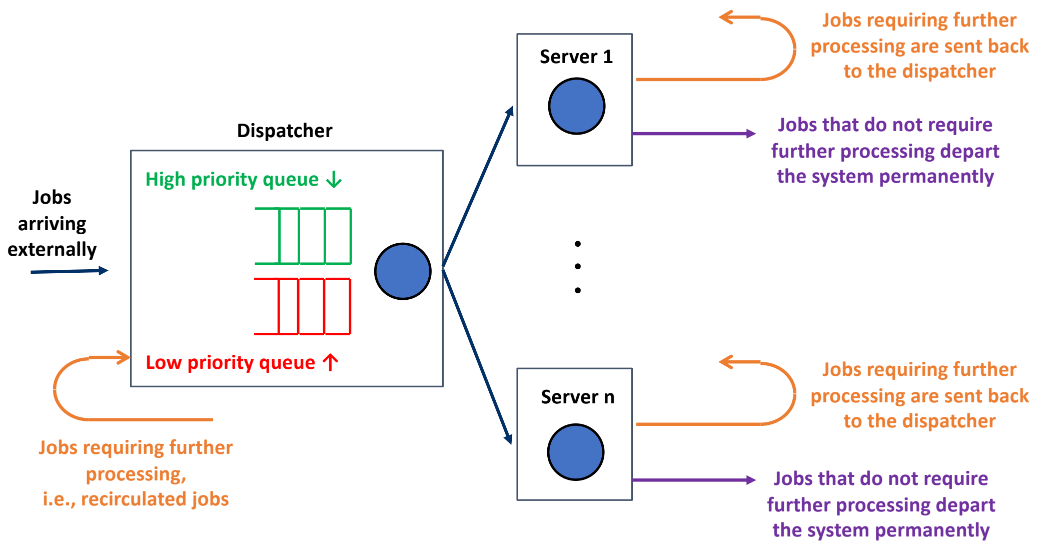
Jobs requiring further processing are sent back to the dispatcher
Server 1
Jobs that do not require further processing depart the system permanently
Dispatcher
High priority queue ↓
Jobs arriving externally
Low priority queue ↑
Jobs requiring further processing, i.e., recirculated jobs
Server n
Jobs requiring further processing are sent back to the dispatcher
Jobs that do not require further processing depart the system permanently

Figure 1: The system for this project.

2 Support provided and computing resources

If you have problems doing this project, you can post your question on the course forum. **We strongly encourage you to do this as asking questions and trying to answer them is a great way to learn. Do not be afraid that your question may appear to be silly, the other students may very well have the same question!** Please note that if your forum post shows part of your solution or code, you must mark that forum post **private**.

Another way to get help is to attend a consultation (see the Timetable section of the course website for dates and times).

If you need computing resources to run your simulation program, you can do it on the VLAB remote computing facility provided by the School. Information on VLAB is available here: `https://taggi.cse.unsw.edu.au/Vlab/`

3 Multi-server system configuration and job characteristics for this project

The configuration of the system that you will use in this project is shown in Figure 1. The system consists of a dispatcher and n servers where $n > 1$. The dispatcher has two queues: a high priority queue and a low priority queue. You can assume that both queues have infinite queueing slots. You have not learnt about priority queues yet but the following description will explain how priority queues are used.

We will use the word *job* to refer to a request that requires service from this system. A job may require one or more visits to the servers in order to get all its work completed. These visits of a job take place one after another with a possible time gap between two consecutive visits. Jobs in this system do not use parallel processing so each job does not use more than one server at a time.

We will now explain how this system handles a new job. When a new job (i.e., an external arrival) arrives at the system, the dispatcher will send the job to any one of the idle servers if there is at least one idle server. If all the servers are busy, the dispatcher will place this job at the end of the high priority queue.

After a job has completed a visit to the server, the job either requires or does not require further visits to the servers. If the job does not require further visits to the servers, then the job will depart from the system permanently. If the job requires further visits to the servers, then the job will be sent back to the dispatcher. We will use the term *re-circulated jobs* to refer to those jobs that are sent back to the dispatcher from the servers because these jobs require further visits to the servers.

A job that arrives at the dispatcher can either be a new job or a re-circulated job, see Figure 1. We have already explained how the dispatcher handles new jobs. We will start to describe how the dispatcher handles the re-circulated jobs. Since the dispatcher handles all re-circulated jobs in the same way, the procedure therefore applies to a generic re-circulated job. We first need to define some notation. First, when a re-circulated job arrives at the dispatcher, the job can have completed 1, 2, 3 or more visits to the servers. We will use c to denote the number of completed server visits when a re-circulated job arrives at the dispatcher. Second, the dispatcher uses a threshold h, which is an integer bigger than or equal to 1, to decide on whether an arriving re-circulated job should be considered a high or low priority job. Now we have defined the notation, we can state the rule that the dispatcher uses: When a re-circulated job arrives at the dispatcher, the dispatcher will classify this job as *low* priority if its value of c is greater than or equal to h; otherwise the job is a *high* priority job. Let us consider an example.

Example 1 *In this example, we assume the threshold h has a value of 2. Let us consider a job which requires altogether 3 server visits before it will permanently depart from the system. So, this job will re-circulate to the dispatcher two times: once with a value of $c = 1$ and the other with $c = 2$.*

- *When this job re-circulates to the dispatcher the first time, its value of c will be 1. Since $c \geq h$ does not hold, the dispatcher will consider this job as a high priority job on this occasion.*
- *The second time that this job re-circulates to the dispatcher, its value of c will be 2. Since $c \geq h$ holds, the dispatcher will consider this job as a low priority job on this occasion.*

We have now explained how the dispatcher classifies an arriving re-circulated job into either a high or low priority job. We have yet to explain the detailed working of the dispatcher. We will do that together with the description of how departures are handled. This is because the arrival of a re-circulated job at a dispatcher follows the job's earlier departure from a server, see Figure 1. The following steps describe how a job, which has completed a server visit, will be handled. For ease of referral, we will use the term *tagged job* to refer to this job that has just completed its server visit.

- The tagged job is considered to be a permanent departure if the number of complete visits that it has already made is equal to the total number of visits that this job requires. If the tagged job is not a permanent departure, then it will be re-circulated to the dispatcher. The server that was working on the tagged job would send a message to the dispatcher to inform it that it is available to serve another job.
- If the tagged job is a re-circulated job, then it will be sent to the dispatcher which will classify it into either a high or low priority job using the values of c and h as described earlier. The dispatcher will then place the tagged job at the end of the appropriate queue.

- The dispatcher is aware that a server has just completed a visit of a job and is available to process another job. The dispatcher executes the following:
  - If the high priority queue is non-empty, then the job at the head of the high priority queue will be sent to the available server for processing.
  - If the high priority queue is empty and the low priority queue is non-empty, then the job at the head of the low priority queue will be sent to the available server for processing.
  - If both high and low priority queues are empty, then the dispatcher does not need to do anything. The server that has just been made available will go idle.

We remark that the above description means that the dispatcher uses the non-preemptive queueing discipline. We will be discussing queueing disciplines in Week 7 and you can read about it on p. 500 of [1]. However, the above description should be enough for you to get your project going now even before we discuss priority queues in Week 7.

We make the following assumptions on the system in Figure 1. First, it takes the dispatcher negligible time to process a job, to classify a job and to send a job to an available server. Second, it takes a negligible time for a server to send a re-circulated job to the dispatcher and to inform the dispatcher on its availability. As a consequence of these assumptions, it means that: (1) If a job arriving at the dispatcher is to be sent to an available server right away, then its arrival time at the dispatcher is the same as its arrival time at the chosen server; (2) The departure time of a job from the dispatcher is the same as its arrival time at the chosen server; and (3) The departure time of a re-circulated job from a server is the same as its arrival time at the dispatcher. Ultimately, these assumptions imply that the response time of the system depends only on the queues and the servers.

We have now completed our description of the operation of the system in Figure 1. We will provide a number of numerical examples to further explain its operation in Section 4.

You will see from the numerical examples in Section 4 that the threshold h can be used to influence the system's mean response time. So, a design problem that you will consider in this project is to determine the value of the threshold h to minimise the mean response time of the system. You can read in [1] how priority queueing can be used to reduce the mean response time of computer systems.

Remark 1 *This project is inspired by a recent work [2] which studies how priority queueing can be used to improve the performance of a multi-server system that provide service to multi-phase jobs. A multi-phase job also requires multiple visit to the servers in order to get its work done. However, the multi-phase job in [2] will sometimes require only the service of a server but sometimes it may require a number of servers in parallel. In order to make this project more do-able, we have simplified many of the settings in [2]. For example, we do not use preemptive queueing, processor sharing and parallel servers.*

4 Examples

We will now present two examples to illustrate the operation of the system that you will simulate in this project. In all these examples, we assume that the system is initially empty.

4.1 Example 1: number of servers $n = 2$ and threshold $h = 1$

In this example, we assume the there are $n = 2$ servers in the system and the threshold h for determining whether a re-circulated job is of low or high priority is 1.

In this example, each job requires one or two visits to the servers before it permanently departs from the system. Table 1 shows, for each job, its arrival time and the service times for its visits. If there is only one service time in the third column in Table 1, then it means the job only requires one server visit. If there are two service times, then the job requires two server visits. For example, Job 1 in Table 1 requires two visits where the first and second visits require, respectively, 3 and 10 time units of service times. As another example, Job 3 requires only one visit and the service time required for that visit is 6 time units.

Job index	Arrival time	Service times of the job's server visits
1	0.9	3, 10
2	1.5	2, 1
3	2.2	6
4	3.3	2
5	8.0	1, 4

Table 1: Data for Example 1.

In this example, a job will be identified with using the tuple $(i, c/r)$ where i is the job's index (see the first column of Table 1), c is the number of complete servers visits made by the job and r is the total number of server visits required by the job. For example,

- The job $(1, 0/2)$ refers to the job with index 1. We know from Table 1 that Job 1 requires 2 visits to the servers and this is indicated by "/2". The notation "0/2" says that this job has done zero complete visits to the servers. When Job 1 re-circulates to the dispatcher for the first time, its tuple becomes $(1, 1/2)$.
- The job $(5, 1/2)$ refers to Job 5 which requires altogether 2 visits to the servers. The notation "1/2" says that this job has done one complete visit to the servers out of the two required visits.

Remark 2 *We remark that the job indices are not necessary for carrying out the discrete event simulation. We have included the job index to make it easier to refer to a job in our description below.*

The events in the system in Figure 1 are the arrival of a *new job* to the dispatcher and the completion of a visit at a server. Note that we have not included the arrival of a *re-circulated* job to the dispatcher as an event. This is because the arrival of a re-circulated job at the dispatcher is *immediately after* the completion of a server visit. So the simulation will handle the arrival of re-circulated job at the dispatcher and its associated server completion together.

We will illustrate how the simulation of the system works using "on-paper simulation". The quantities that you need to keep track of are:

- **Next arrival time** is the time that the next *new job* will arrive
- For each server, we keep track its server status, which can be busy or idle.
- We also keep track of the following information on the job that is being processed in the server:
  - **Next completion time** is the time at which the job will complete its current server visit. If the server is idle, the next completion time is set to ∞. Note that there is a next completion time for each server.
  - The time that this job arrived at the system. This is needed for calculating the response time of the job when it permanently departs from the system.

  - A list of the service times for the future server visits of this job. Note that we enclose the list of service times within a pair of square brackets [].
  - The job's tuple.

  For example, the job information "3.5, 1.5, [1], (2,0/2)" indicates that current visit will be completed at time 3.5 and this job arrived at the system at time 1.5. The "0/2" indicates that the job has not completed any server visits so the current visit is the job's first visit to the server. The "[1]" indicates that the job needs one more visit in the future and this visit will require a service time of 1. Note that if the job has no more future visits to make, then we will use [ ] to indicate that.

- The contents of the high and low priority queues. Each job in the queue is identified by 3 fields: the job's tuple, the job's arrival time to the system, a list of the job's service times for its future server visits. For example, we write a job in a queue as

  [(1,1/2), 0.9, [10]]

  which means the job (1,1/2) arrived at the system at time 0.9, has 1 visit completed and its future visit to the server will require a service time of 10.

The "on-paper simulation" is shown in Table 2. The notes in the last column explain what updates you need to do for each event. Recall that the two event types in this simulation are the arrival of a *new job* to the dispatcher and the completion of a visit at a server, we will simply refer to these two events as *Arrival* and *Completion* in the "Event type" column (i.e., second column) in Table 2.

Master clock	Event type	Next arrival time	Server 1	Server 2	High priority queue	Low priority queue	Notes
0	–	0.9	Idle, ∞	Idle, ∞	–	–	We assume the servers are idle and queues are empty at the start of the simulation. The next departure times for all servers are ∞. The "–" indicates that the queues are empty.
0.9	Arrival	1.5	Busy, 3.9, 0.9, [10], (1,0/2)	Idle, ∞	–	–	This event is the arrival of Job 1 as a new job. Since all the servers were idle before this arrival, the job can be sent to any one of the idle servers. We have chosen to send this job to Server 1. At the time this job enters the server, the job has zero completed visits so it is identified by (1, 0/2). The job requires a service time of 3 for *this* visit and it starts to receive service at time 0.9, so its completion time is 3.9. Note that the record of the job in the server also includes the list of service times [10] that the job requires in its future visit; you will see how this information will be used later on. From Table 1, you can see that Job 1 requires two visits with service times 3 and 10, and we want to point out we no longer need to remember the first service time 3 as it is for this visit and will not be needed again in the future; the service time 3 has been used to compute the completion time of this visit and this is all we need this service time for. Lastly, we need to update the arrival time of the next new job, which is 1.5.
1.5	Arrival	2.2	Busy, 3.9, 0.9, [10] (1,0/2)	Busy, 3.5, 1.5, [1], (2,0/2)	–	–	This event is the arrival of Job 2 as a new job. Since there was an idle server before this arrival, the job can be sent to the idle server. The job (2, 0/2) requires a service time of 2 for this visit and it starts to receive service at time 1.5, so the completion time of this visit is 3.5. Lastly, we need to update the arrival time of the next new job, which is 2.2.

2.2	Arrival	3.3	Busy, 3.9, 0.9, [10], (1,0/2)	Busy, 3.5, 1.5, [1] (2,0/2)	[(3,0/1), 2.2, [6]]	–	This event is the arrival of Job 3 as a new job. Since both servers were busy before this arrival, the job is placed at the end of the high priority queue. The record of the job in the queue has 3 components: (i) Its tuple identifier (3, 0/1) indicating that it has not received any service; (ii) Its arrival time at 2.2; (iii) A list of its service times for its future visits and for this job, this is [6] as it requires 6 time units for a future server visit. The number 6 will be used to determine the completion time when this job enters into a server. Lastly, we need to update the arrival time of the next new job, which is 3.3.
3.3	Arrival	8.0	Busy, 3.9, 0.9, [10], (1,0/2)	Busy, 3.5, 1.5, [1] (2,0/2)	[(3,0/1), 2.2, [6]], [(4,0/1), 3.3, [2]]	–	This event is the arrival of Job 4 as a new job. Since both servers were busy before this arrival, the job is placed at the end of the high priority queue. Lastly, we need to update the arrival time of the next new job, which is 8.
3.5	Completion	8.0	Busy, 3.9, 0.9, [10], (1,0/2)	Busy, 9.5, 2.2, [], (3,0/1)	[(4,0/1), 3.3, [2]]	[(2,1/2), 1.5, [1]]	This event is the completion of a visit of the Job (2,0/2). Upon this completed visit, this job will now be identified by (2,1/2). The identifier indicates that this job will require future visits to the server so this job will be re-circulated to the dispatcher. The dispatcher sees that this job has $c = 1$ and finds that $c \geq h$ holds, so this job should join the low priority queue. Since the high priority queue is non-empty, the job at the top of the priority queue will move to the server. This job requires 6 units of service so its completion time should be $3.5 + 6 = 9.5$. Note that the list of future service times in the server for this job is now empty (i.e, []) rather than [6] because we have already used the number 6 to calculate the job's completion time for this visit, so this number will no longer be needed in the future.

3.9	Completion	8.0	Busy, 5.9, 3.3, [], (4,0/1)	Busy, 9.5, 2.2, [], (3,0/1)	–	[(2,1/2),1.5,[1]], [(1,1/2),0.9, [10]]	This event is the completion of a visit of Job (1,0/2). Upon this completed visit, this job will now be identified by (1,1/2). This job will be re-circulated to the dispatcher. Since $c = 1$, the dispatcher will place this job at the end of the low-priority queue. Since the high priority queue is non-empty, the job at the top of the priority queue (4,0/1) will move to the server. This job requires 2 units of service so its completion time should be $3.9 + 2 = 5.9$.
5.9	Completion	8.0	Busy, 6.9, 1.5, [], (2,1/2)	Busy, 9.5, 2.2, [], (3,0/1)	–	[(1,1/2),0.9, [10]]	This event is the completion of a visit of Job (4,0/1). Since Job 4 requires only one visit in total, this job will permanently depart from the system. The response time of Job 4 is therefore its departure time (=5.9) minus its arrival time (=3.3), i.e. response time is 5.9–3.3 = 2.6. Since Server 1 is now available, the dispatcher will need to check the status of the queues. Since the high priority queue is empty and the low priority is non-empty, the job at the top of the low priority queue will head to the server. The scheduled completion time of the job in Server 1 is 5.9+1=6.9.
6.9	Completion	8.0	Busy, 16.9, 0.9, [], (1,1/2)	Busy, 9.5, 2.2, [], (3,0/1)	–	–	This event is the completion of a visit of Job (2,1/2). Since Job 2 requires two server visits in total and both visits have now been completed, so this job will permanently depart from the system. The response time of Job 2 is therefore its departure time (=6.9) minus its arrival time (=1.5), i.e. response time is 6.9–1.5 = 5.4. With this departure, Server 1 is now available. Since the high priority queue is empty and the low priority is non-empty, the job at the top of the low priority queue will head to the server. The scheduled completion time of the job in Server 1 is 6.9+10=16.9.
8.0	Arrival	∞	Busy, 16.9, 0.9, [], (1,1/2)	Busy, 9.5, 2.2, [], (3,0/1)	[(5,0/2), 8, [1,4]]	–	This event is the arrival of Job 5 as a new job. Since both servers were busy before this arrival, the job is placed at the end of the high priority queue. Since there are no more jobs after this arrival, the next arrival time is set to ∞.

9.5	Completion	∞	Busy, 16.9, 0.9, [], (1,1/2)	Busy, 10.5, 8.0, [4], (5,0/2)	–	–	This event is the completion of a visit of Job (3,0/1). Since Job 3 requires only one visit in total, this job will permanently depart from the system. The response time of Job 3 is therefore its departure time (=9.5) minus its arrival time (=2.2), i.e. response time is 9.5–2.2 = 7.3. Server 2 is now available. Since the high priority queue is non-empty, the job (5, 0/2) at the top of the high priority queue will head to Server 2. The scheduled completion time of this job in Server 2 is 9.5+1=10.5.
10.5	Completion	∞	Busy, 16.9, 0.9, [], (1,1/2)	Busy, 14.5, 8.0, [], (5,1/2)	–	–	This event is the completion of a visit of Job (5,0/2). Upon the completion of this visit, the job becomes (5,1/2) and will be recirculated to the dispatcher. Since $c \geq h$ holds for this job, it is classified as a low priority job and is placed at the end of the low priority queue. Server 2 is now available. Since the high priority queue is empty and the job at the head of the low priority queue is in fact (5,1/2), so this job heads to Server 2. The scheduled completion time of the job in Server 2 is 10.5+4=14.5.
14.5	Completion	∞	Busy, 16.9, 0.9, [], (1,1/2)	Idle, ∞	–	–	This event is the completion of a visit of Job (5,1/2). Since Job 5 requires two server visits in total and both visits have now been completed, so this job will permanently depart from the system. The response time of Job 5 is therefore its departure time (=14.5) minus its arrival time (=8), i.e. response time is 14.5–8 = 6.5. Since both queues are empty, Server 2 becomes idle and its next departure time is set to ∞.
16.9	Completion	∞	Idle, ∞	Idle, ∞	–	–	This event is the completion of a visit of Job (1,1/2). Since Job 1 requires two server visits in total and both visits have now been completed, so this job will permanently depart from the system. The response time of Job 1 is therefore its departure time (=16.9) minus its arrival time (=0.9), i.e. response time is 16.9–0.9 = 16. Since both queues are empty, Server 1 becomes idle and its next departure time is set to ∞. Since there are no more arrivals and departures, the simulation is now completed.

Table 2: "On paper simulation" illustrating the event updates of the system.

The above description has not explained what happens if an arrival event and a completion event are at the same time. We will leave it unspecified. If we ask you to simulate in trace driven mode, we will ensure that such situation will not occur. If the inter-arrival time and service time are generated randomly, the chance of this situation occurring is practically zero so you do not have to worry about it.

Table 3 summarises the arrival, departure and response times of the jobs in this example. The mean response time is the sum of the last column in the table divided by the number of jobs that have permanently departed from the system, which is $\frac{37.8}{5} = 7.56$.

Table 4 shows the times at which the server visits are completed. These are also the times for the completion events.

Job	Arrival time	Departure time	Response time
1	0.9	16.9	16.0
2	1.5	6.9	5.4
3	2.2	9.5	7.3
4	3.3	5.9	2.6
5	8.0	14.5	6.5

Table 3: The arrival and departure times of the jobs in Example 1.

Arrival time	Completion time	Number of completed server visits
1.5	3.5	1
0.9	3.9	1
3.3	5.9	1
1.5	6.9	2
2.2	9.5	1
8.0	10.5	1
8.0	14.5	2
0.9	16.9	2

Table 4: The completion times for the server visits.

4.2 Example 2: number of servers $n = 2$, threshold $h = 2$

This example is identical to Example 1 except that the threshold $h = 2$. Since all the jobs in Example 1 require at most two visits, therefore all the re-circulated jobs will go into the high priority queue and the low priority queue is not used at all.

For this value of $h = 2$, the departure times of the jobs are the same as the case for $h = 1$. The server visit completion times are also the same as those in Table 4.

Note that Tables 5 and 6 in Version 1.00 are incorrect, and they have been removed from Version 1.01. There are no Tables 5 and 6 in this document. The next table number is 7.

4.3 Example 3: number of servers $n = 3$, threshold $h = 1$

This example is based on the arrivals and service times in Table 7. The number of servers n is 3 and threshold $h = 1$.

Table 8 summarises the arrival and departure times of all the jobs. The mean response time of the 4 jobs in this example is $\frac{52.6}{4} = 13.15$.

Table 9 shows the times at which the server visits are completed. These are also the times for the completion events.

Job index	Arrival time	Service times of the job's server visits
1	1.0	1.7, 2.8, 2.1
2	1.9	4.1, 4.9, 6.1
3	2.5	12.0
4	3.2	14.0

Table 7: Data for Example 3.

Job	Arrival time	Departure time	Response time
1	1.0	8.1	7.1
2	1.9	19.1	17.2
3	2.5	14.5	12.0
4	3.2	19.5	16.3

Table 8: The arrival and departure times of the jobs for Example 3.

Arrival time	Completion time	Number of completed server visits
1.0	2.7	1
1.0	5.5	2
1.9	6.0	1
1.0	8.1	3
1.9	13.0	2
2.5	14.5	1
1.9	19.1	3
3.2	19.5	1

Table 9: The completion times for the server visits.

4.4 Example 4: number of servers $n = 3$, threshold $h = 2$

This example is identical to Example 3 except that the threshold $h = 2$.

Table 10 summarises the arrival and departure times of all the jobs. The mean response time of the 4 jobs in this example is $\frac{57.5}{4} = 14.375$, which is higher than that of Example 3. This demonstrates that priority queueing can sometimes be used to influence the mean response time of the system. However, as demonstrated in Section 4.2, sometimes h does not seem to change the response time.

Table 11 shows the times at which the server visits are completed. These are also the times for the completion events.

Job	Arrival time	Departure time	Response time
1	1.0	13.0	12.0
2	1.9	19.1	17.2
3	2.5	14.5	12.0
4	3.2	19.5	16.3

Table 10: The arrival and departure times of the jobs for Example 4.

Arrival time	Completion time	Number of completed server visits
1.0	2.7	1
1.0	5.5	2
1.9	6.0	1
1.9	10.9	2
1.0	13.0	3
2.5	14.5	1
1.9	19.1	3
3.2	19.5	1

Table 11: The completion times for the server visits.

5 Project description

This project consists of two main parts. The first part is to develop a simulation program for the system in Fig. 1. The system has already been described in Section 3 and illustrated in Section 4. In the second part, you will use the simulation program that you have developed to solve a design problem.

5.1 Simulation program

You must write your simulation program in one (or a combination) of the following languages: Python 3 (note: version 3 only), C, C++, or Java. All these languages are available on the CSE system.

We will test your program on the CSE system so your submitted program **must** be able to run on a CSE computer. Note that it is possible that due to version and/or operating system differences, code that runs on your own computer may not work on the CSE system. It is your responsibility to ensure that your code works on the CSE system.

Note that our description uses the following variable names:

1. A variable `mode` of string type. This variable is to control whether your program will run simulation using randomly generated arrival times and service times; or in trace driven mode. The value that the parameter `mode` can take is either `random` or `trace`.
2. A variable `time_end` which stops the simulation if the master clock exceeds this value. This variable is only relevant when `mode` is `random`. This variable is a positive floating point number.

Note that your simulation program must be a general program which allows different parameter values to be used. When we test your program, we will vary the parameter values. You can assume that we will only use valid inputs for testing.

For the simulation, you can always assume that the system is empty initially.

5.1.1 The random mode

When your simulation is working in the `random` mode, it will generate the **inter-arrival** times and the workload of a job in the following manner.

1. We use $\{a_1, a_2, \ldots, a_k, \ldots, ...\}$ to denote the inter-arrival times of the jobs arriving at the dispatcher. These inter-arrival times have the following properties:
   (a) Each a_k is the product of two random numbers a_{1k} and a_{2k}, i.e $a_k = a_{1k}a_{2k}$ $\forall k = 1, 2, ...$
   (b) The sequence a_{1k} is exponentially distributed with a mean arrival rate λ requests/s.
   (c) The sequence a_{2k} is uniformly distributed in the interval $[a_{2l}, a_{2u}]$.

   Note: The easiest way to generate the inter-arrival times is to multiply an exponentially distributed random number with the given rate and a uniformly distributed random number in the given range. It would be more difficult to use the inverse transform method in this case, though it is doable.

2. The workload of a job is characterised by; (i) the number of server visits that the job requires; and (ii) the service times of all these server visits.

   The first step to generate the workload of a job is to generate a random positive integer to use as the number of server visits that this job requires. You will be given a sequence of J

non-negative real numbers p_1, p_2, ..., p_k, ... and p_J with the property $\sum_{k=1}^{J} p_k = 1$. Given these numbers, we want the probability that a job needs k server visits to be equal to p_k, for $k = 1, ..., J$.

For example, if you are given the sequence 0.5, 0.2, 0.3, then the jobs generated has the following properties:

(a) Prob[a job requires 1 server visit] = 0.5

(b) Prob[a job requires 2 server visits] = 0.2

(c) Prob[a job requires 3 server visits] = 0.3

Note that you may interpret J as the maximum number of server visits that a generated job requires. In the example above, we have $J = 3$, which implies that all generated jobs need at most 3 server visits.

3. If a job requires k server visits, then you will need to generate k random service times for each of the k server visits. These k service times are independent and they all come from the same probability distribution.

   The service time per server visit is generated by the probability density function (PDF) $g(t)$ where:

$$g(t) = \begin{cases} 0 & \text{for } 0 \leq t \leq \alpha \\ \frac{\gamma}{t^\beta} & \text{for } \alpha < t \end{cases} \tag{1}$$

   where

$$\gamma = \frac{\beta - 1}{\alpha^{1-\beta}}$$

   Note that this probability density function has two parameters: α and β. You can assume that $\alpha > 0$ and $\beta > 3$.

   As an example, if a job requires 3 server visits, then you will need to generate 3 random numbers which come from the probability distribution whose PDF is given by $g(t)$.

5.1.2 The trace mode

When your simulation is working in the `trace` mode, it will read the list of **inter-arrival** times and the list of service times of the server visits from two separate ASCII files. We will explain the format of these files in Sections 6.1.3 and 6.1.4 .

An **important requirement** for the `trace` mode is that your program is required to simulate until all jobs have departed from the system. You can refer to Table 2 for an illustration.

Hint: Do **not** write two separate programs for the `random` and `trace` modes because they share a lot in common. A few `if–else` statements at the right places are what you need to have both modes in one program.

5.2 Determining the threshold h that minimises the mean response time

After writing your simulation program, your next step is to use your simulation program to determine the threshold h that can minimise the mean response time.

For this design problem, you will assume the following parameter values:

- Number of servers: $n = 6$

- For inter-arrival times: $\lambda =$ ~~3.9~~ 3.1, $a_{2\ell} = 0.91$, $a_{2u} = 1.27$
- For the number of server visits required for each job: the sequence p_1, p_2, p_3, p_4, p_5 is ~~0.52, 0.21, 0.15, 0.08, 0.04~~ 0.32, 0.21, 0.15, 0.08, 0.24.
- For the service time per server visit: $\beta = 3.4$, $\alpha = 0.3$.

In solving this design problem, you need to ensure that you use **statistically sound** methods to compare systems. You will need to consider simulation controls such as length of simulation, number of replications, transient removals and so on. You will need to justify in your report on how you determine the value of the threshold h.

6 Testing your simulation program

In order for us to test the correctness of your simulation program, we will run your program using a number of test cases. The aim of this section is to describe the expected input/output file format and how the testing will be performed.

Each test is specified by 4 configurations files. We will index the tests from 1. If 12 tests are used, then the indices for the tests are 1, 2, ..., 12. The names of the configuration files are:

- For Test 1, the configuration files are `mode_1.txt`, `para_1.txt`, `interarrival_1.txt` and `service_1.txt`. The files are similarly named for indices 2, 3, .., 9.
- For Test 10, the configuration files are `mode_10.txt`, `para_10.txt`, `interarrival_10.txt` and `service_10.txt`. The files are similarly named if the test index is a 2-digit number.

We will refer to these files using the generic names `mode_*.txt`, `para_*.txt` etc. We will describe the format of the configuration files in Section 6.1

Each test should produce 2 output files whose format will be described in Section 6.2. We will explain how testing will be conducted in Sections 6.3 and 6.5.

6.1 Configuration file format

Note that Test 1 is the same as Example 1 discussed in Section 4.1. We will use that test to illustrate the file format.

6.1.1 `mode_*.txt`

This file is to indicate whether the simulation should run in the `random` or `trace` mode. The file contains one string, which can either be `random` or `trace`.

6.1.2 `para_*.txt`

If the simulation mode is `trace`, then this file has two lines. The first line is the value of n (= number of servers) and the second line has the value of h (= threshold for priority queueing). If the test is Example 1 in Section 4.1, then the contents of this file are:

```
2
1
```

These values are in the sample file `para_1.txt`.

If the simulation mode is `random`, then the file has three lines. The meaning of the first two lines are the same as above. The last line contains the value of `time_end`, which is the end time of the simulation. The contents of the sample file `para_7.txt` are shown below where the last line indicates that the simulation should run until `1000`.

```
3
1
1000
```

You can assume that we will only give you valid values. You can expect n to be a positive integer greater than 1. You can expect h to be a positive integer. For `time_end`, it is a strictly positive integer or floating point number.

6.1.3 `interarrival_*.txt`

The contents of the file `interarrival_*.txt` depend on the `mode` of the test. If mode is `trace`, then the file `interarrival_*.txt` contains the interarrival times of the jobs with one interarrival time occupying one line. You can assume that the list of interarrival times is always positive. For Example 1 in Section 4.1, the arrival times are $[0.9, 1.5, 2.2, 3.3, 8.0]$ which means the inter-arrival times are $[0.9, 0.6, 0.7, 1.1, 4.7]$. For this example, the inter-arrival times will be specified by a file (see sample file `interarrival_1.txt`) whose contents are:

```
0.9000
0.6000
0.7000
1.1000
4.7000
```

If the mode is `random`, then the file `interarrival_*.txt` contain 2 lines. The first line contains three values corresponding to the parameters λ, $a_{2\ell}$ and a_{2u}. The second line contains the the values for the sequence $p_1, ..., p_J$. As an example, the contents of `interarrival_8.txt` are:

```
1.0 0.95 1.2
0.5 0.3 0.15 0.05
```

For this example, the values of λ, $a_{2\ell}$ and a_{2u} are respectively 1.0, 0.95 and 1.2. The values of p_1, p_2, p_3, p_4 are 0.5, 0.3, 0.15, 0.05. This means that you can infer the value of J by counting the number of values found in the second line of the file. For `interarrival_8.txt`, $J = 4$. Note that you can assume that we will only give you valid p_k, i.e. all p_k's are non-negative and the sum of all p_k's is 1.

6.1.4 `service_*.txt`

For `trace` mode, the file `service_*.txt` contains the service times of the server visits. As an illustration, the service times of the server visits for Example 1 in Section 4.1 will be specified by a file (see sample file `service_1.txt`) whose contents are:

```
3.0000 10.0000
2.0000 1.0000
6.0000 NaN
2.0000 NaN
1.0000 4.0000
```

where you will find the service times of the server visits of each job in a line of the file.

Note that the symbol `NaN` is a Python floating point number to denote *not a number* and is often used to indicate an absence of numbers. In this example, if there are two numbers on the line, the job requires two server visits; if there is a number and an `NaN`, the job is requires only one server visit.

~~The following shows the contents of service_3.txt for trace mode simulation:~~ The following shows what the contents of a file for service times may look like for the trace mode simulation mode:

```
2.1000 3.2000 1.9000 NaN
4.0000 3.0000 4.9000 6.1000
5.1000 2.3000 1.2000 NaN
7.2000 1.8000 NaN NaN
4.6000 NaN NaN NaN
```

Note that there are 4 entries in each line where the number 4 corresponds to the maximum number of server visits among all the jobs. You can conveniently load the contents of this file by using the function `numpy.loadtxt()` into a `numpy` array. You may also find the function `numpy.isnan()` useful.

In general, if the maximum number of server visits among all jobs is V, then there are V entries in each line of `service_*.txt`.

For `random` mode, the file `service_*.txt` contains one line, corresponding to the values of β and α.

You can assume that the data we provide for `trace` mode are consistent in the following way: The number of inter-arrival times and the number of lines of service times are equal.

6.2 Output file format

In order to test your simulation program, we need two output files **per test**. One file containing the mean response time. The other file contains the completion times of the *server visits* from the *servers*.

We want to start by clarifying what we mean by mean response time. You can calculate the response time of a job by subtracting the time that this job arrives at the system as a new job from the time it permanently departs from the system. Tables 1 and **??** illustrate this concept.

For `trace` mode, the mean response time will be calculated using all the jobs provided in the `interarrival_*.txt` and `service_*.txt`. This is because, as mentioned in Section 5.1.2, a `trace` mode simulation is required to simulate until all jobs have permanently departed from the system.

For `random` mode, the mean response time should be calculated using all those jobs that have permanently departed the system by `time_end`. In other words, for those jobs which are still in the queue or are being processed in the server at `time_end`, you do not include these jobs when calculating the mean response time.

Note that you do not have to consider transient removal for the mean response before you write the result to the output file. However, you should consider transient removal when you do your design.

The mean response time should be written to a file whose filename has the form `mrt_*.txt`. For Example 1 in Section 4.1, the expected contents of this file are:

```
7.5600
```

The other file `dep_*.txt` contains the completion times of of the *server visits* from the *servers*. For Example 1 in Section 4.1, the expected contents of this file are:

```
1.5000 3.5000 1 2
0.9000 3.9000 1 2
3.3000 5.9000 1 1
```

```
1.5000 6.9000 2 2
2.2000 9.5000 1 1
8.0000 10.5000 1 2
8.0000 14.5000 2 2
0.9000 16.9000 2 2
```

Note the following requirements for the file containing the completion times:

1. Each line contains 4 entries.
2. Each line provides the information on the completion time of a server visit.
3. For each line, the first entry is the arrival time of the job to the system (i.e., as a new job), the fourth entry is the total number of server visits required by this job, the third entry is the number of complete server visits that this job has made at the time given by the second entry. Let us take the first line `1.5000 3.5000 1 2`. It says that the job that arrives at the system at time 1.5 requires a total of 2 server visits, and at time 3.5, this job has completed 1 server visit. You should be able to reconcile the contents of the above file with Example 1 in Section 4.1.
4. The server visits must be ordered according to *ascending* completion times.
5. If the simulation is in the `trace` mode, we expect the simulation to finish after all jobs have been processed. Therefore, the number of lines in `dep_*.txt` should be equal to the total number of server visits of all jobs.
6. If the simulation is in the `random` mode, the file should contain all the server visits that have been completed by `time_end`.

All mean response times, arrival times and completion times in `mrt_*.txt` and `dep_*.txt` should be printed as floating point numbers to exactly 4 decimal places. Note that your simulation should be performed in full floating point precision and you should only do the rounding when you are writing the output files.

6.3 The testing framework

When you submit your project, you must include a Linux bash shell script with the name `run_test.sh` so that we can run your program on the CSE system. This shell script is required because you are allowed to use a computer language of your choice.

Let us first recall that each test is specified by a four configuration files and should produce two output files. For example, test number 1 is specified by the configuration files `mode_1.txt`, `interarrival_1.txt`, `service_1.txt` and `para_1.txt`; and test number 1 is expected to produce the output files `mrt_1.txt` and `dep_1.txt`.

We will use the following directory structure when we do testing.

```
the directory containing run_test.sh
├── config/
└── output/
```

We will put all the configuration files for all the tests in the sub-directory `config/`. You should write all the output files to the sub-directory `output/`.

To run test number 1, we use the shell command:

```
./run_test.sh 1
```

The expected behaviour is that your simulation program will read in the configuration files for test number 1 from `config/`, carry out the simulation and create the output files in `output/`.

Similarly, to run test number 2, we use the shell command:

```
./run_test.sh 2
```

This means that the shell script `run_test.sh` has one input argument which is the test number to be used.

Let us for the time being assume that you use Python (Version 3) to write your simulation program and you call your simulation program `main.py`. If the file `main.py` is in the same directory as `run_test.sh`, then `run_test.sh` can be the following one-line shell script:

```
python3 main.py $1
```

The shell script will pass the test number (which is in the input argument `$1`) to your simulation program `main.py`. This also implies that your simulation program should accept one input argument which is the test number.

Just in case you are not familiar with shell script, we have provided two sample files: `run_test.sh` and `main.py` to illustrate the interaction between a shell script and a Python (Version 3) file. You need to make sure `run_test.sh` is executable. (You can make the shell script `run_test.sh`executable by using the command "`chmod u+x run_test.sh`".) If you run the command `./run_test.sh 2`, it will produce a file with the name `dummy_2.txt` in the directory `output/`. You can also try using other input arguments for the sample shell script. You can use these sample files to help you to develop your code.

If you use C, C++ or Java, then your `run_test.sh` should first compile the source code and then run the executable. You should of course pass the test number to the executable as an input.

You can put your code in the same directory that contains `run_test.sh` or in a subdirectory below it. For example, you may have a subdirectory `src/` for your code like the following:

```
the directory containing run_test.sh
├── config/
├── output/
└── src/
```

6.4 Sample files

You should download the file `sample_project_files_25Mar.zip` from the project page on the course website. The zip archive has the following directory structure:

```
Base directory containing cf_output_with_ref.py, run_test.sh and main.py
├── config/
├── output/
└── ref/
```

Details on the zip-archive are:

- The sub-directory `config/` contains configuration files that you can use for testing.
  - The files `mode_1.txt`, `mode_2.txt`, ..., `mode_9.txt` and `mode_10.txt`. Note that Tests 1–6 are for `trace` mode while Tests 7–10 are for `random` mode.
  - The files `para_*.txt`, `interarrival_*.txt` and `service_*.txt` for `*` from 1 to 10, as the input to the simulation.

- Note that Tests 1 and 2 are the same as respectively, Example 1 and Example 2, in Section 4. Also Tests 3 and 4 are the same as Examples 3 and 4 in Section 4.

- The sub-directory `output/` is empty. Your simulation program should place the output files in this sub-dirrectory.
- The sub-directory `ref/` contains the expected simulation results.
  - The files `mrt_*_ref.txt` and `dep_*_ref.txt` for `*` from 1 to 10, as the reference files for the output. For Tests 1–6, you should be able to reproduce the results in `mrt_*_ref.txt` and `dep_*_ref.txt`. However, since Tests 7–10 are in `random` mode, you will not be able to reproduce the results in the output files. They have been provided so that you can check the expected format of the files.
- The Python file `cf_output_with_ref.py` which illustrates how we will compare your output against the reference output. This file takes in one input argument, which is the test number. For example, if you want to check your simulation outputs for test 2, you use:

  ```
  python3 cf_output_with_ref.py 2
  ```

  Note the following:

  - The file `cf_output_with_ref.py` expects the directory structure shown earlier.
  - For `trace` mode, we will check your mean response time and the completion times. Note that we are not looking for an exact match but rather whether your results are within a valid tolerance. The tolerance for the `trace` mode is 10^{-3} which is fairly generous for numbers with 4 decimal places.
  - For `random` mode, we will only check the mean response time. You can see from the sample file that we check whether the mean response time is within an interval. We obtain this interval using the following method: (i) we first simulate the system many times; (ii) we then use the simulation results to estimate the maximum and minimum mean response times; (iii) we use the estimated maximum and minimum values to form an interval; (iv) in order to provide some tolerance due to randomness, we enlarge this interval further.
  - Note that we use a very generous tolerance so if your mean response time does not pass the test, then it is highly likely that your simulation program is not correct.
- The files `run_test.sh` and `main.py` as mentioned in Section 6.3.

6.5 Carrying out your own testing on the CSE system

It is important for you to note the assumption on directory structure mentioned in Section 6.3. You must ensure your shell script and program files are written with this assumption in mind.

Since we will be testing your work on the CSE system, we strongly advise you to carry out the following on the CSE system before submission.

- Create a new folder in your CSE account and `cd` to that folder. We will refer to this directory as the base directory.
  - Copy your shell script `run_test.sh` and program files to the base directory [1].
  - Copy the `config` and `ref` directories, as well as their contents, to the base directory
  - Create an empty directory `output`

[1] **Remark** : In actual testing, we will copy your submitted `project.zip` (see Section 7.3) to this base directory and unzip it. We expect that `run_test.sh` is in this base directory after unzipping.

- Make sure your shell script is executable by using the command "`chmod u+x run_test.sh`"

- Run your shell script for each test one by one. Make sure that each run produces the appropriate output files for that test in the `output` directory.
- Copy `cf_output_with_ref.py` to the base directory. Run it to compare your output against the reference output.

These steps are the same as those that we will use for testing. It is important to know that we will create an empty `output/` directory before we run your code. This means your code does **NOT** have to create the `output/` directory.

The submission portal will make an attempt to run test number 1 with your submitted files, see Section 7.3.

6.6 Getting started and base code

For this project, we do not require you to write your code from scratch. You are allowed to build your project by using: (i) the sample code from COMP9334; or (ii) the code in the public domain as long as it meets the requirements below.

If you intend to use Python 3 to write your simulation code, the best way to get started is to use the M/M/m simulation code provided with the solution to Week 4B's revision problem and modify from there. Sample code for trace driven simulation is provided with the lecture in Week 4B.

There is also a lot of discrete event simulation code in Python 3, C, C++ and Java in the public domain. You are allowed to use the public domain code as a basis for your project work as long as it meets the following requirements:

1. The code has a clearly identifiable author
2. The code has a date which is before the date that this project document is released.
3. You provide us with an URL of the source code.
4. You clearly state the changes that you have made on the original code to adapt it to the specifications of this project.

If you use any public domain code in your project, your project report **must** include the information to satisfy the above four requirements.

If you would like to use a certain public domain source but you are not sure whether it meets our requirements, you can consult the lecturer on the forum using a private message.

If your project work is based on the COMP9334 sample code, then your report **must** state that the COMP9334 sample code has been used and provide information to satisfy Requirement 4 above.

7 Project requirements

This is an individual project. You are expected to complete this project on your own.

7.1 Submission requirements

Your submission should include the following:

1. A written report
   (a) Only soft copy is required.
   (b) It must be in Acrobat pdf format.
   (c) It must be called "report.pdf".
   (d) The report must include the information required in Section 6.6.
2. Program source code:
   (a) For doing simulation
   (b) The shell script `run_test.sh`, see Section 6.3.
3. Any supporting materials, e.g. logs created by your simulation, scripts that you have written to process the data etc.

The assessment will be based on your submission and running your code on the CSE system. It is important that you submit the right version of the code and make sure that it runs on the CSE system.

It is important that you write a clear and to-the-point report. You need to aware that you are writing the report to the marker (the intended audience of the report) not for yourself. Your report will be assessed primarily based on the quality of the work that you have done. You do not have to include any background materials in your report. You only have to talk about how you do the work and we have provided a set of assessment criteria in Section 7.2 to help you to write your report. In order for you to demonstrate these criteria, your report should refer to your programs, scripts, additional materials so that we are aware of them.

7.2 Assessment criteria

We will assess the quality of your project based on the following criteria:

1. The correctness of your simulation code. For this, we will:
   (a) Test your code using test cases
   (b) Look for evidence in your report that you have verified the correctness of the inter-arrival probability distribution, probability distribution of the number of server visits, and service time distribution. You can include appropriate supporting materials to demonstrate this in your submission.
   (c) Look for evidence in your report that you have verified the correctness of your simulation code. You may derive test cases such as those in Section 4 to test your code. You can include appropriate supporting materials to demonstrate this in your submission.
2. You will need to demonstrate that your results are reproducible. You should provide evidence of this in your report.
3. For the part on determining a suitable value of the threshold h that minimises the mean response time, we will look for the following in your report:
   (a) Evidence of using statistically sound methods to analyse simulation results
   (b) Explanation on how you choose your simulation and data processing parameters, e.g lengths of your simulation, number of replications, end of transient etc.

The above marking criteria closely follow the messages that we have been promoting in our lectures on discrete event simulation. You need to ensure that your simulation code is correct and at the same time you need to consider the choice of simulation parameters and use statistical sound method to compare systems. If you want to do well for the project, you must make sure that you cover all the above aspects.

7.3 How to submit

You should "zip" your report, shell script, programs and supporting materials into a file called "project.zip". The submission system will only accept this filename. **Please ensure that you run `zip` in the directory containing your `run_test.sh` because our test program expects to find `run_test.sh` at certain location, see Footnote 1 on page 22.** If you need to store directories when zipping, you need to use the `-r` switch to preserve the relative path.

You should submit your work via the course website. Note that the maximum size of your submission should be no more than 20MBytes.

You can submit multiple times before the deadline. A later submission overrides the earlier submissions, so make sure you submit the correct file. We will only mark the last submission that you make. Do not leave until the last moment to submit, as there may be technical or communication error and you will not have time to rectify.

When you submit your files, the submission portal will unzip your `project.zip` and run sample test 1. If the portal says that your `run_test.sh` is not at the right location, it probably means that you have not run zip in the directory containing your `run_test.sh`. You can do this test after you have got the simulation part ready and before you attempt the design. Since later submissions will overwrite the earlier ones, you can get this test done earlier.

8 Further project conditions

1. The total mark for this project is 30 marks.
2. The submission deadline is 5:00pm Friday 21 April 2023. Submissions made after the deadline will incur a penalty of 5% per day. The penalty is applied to the mark that you would have received if the submission was not late. Late submissions will only be accepted until 5:00pm Wednesday 26 April 2023, after which no submissions will be accepted.
3. If you use a computer program to perform any part of your work, you **must** submit the program or you lose marks for that component. This requirement applies to computer programs for simulation as well as those for statistical analysis.
4. Additional project conditions:
   - Joint work is not permitted on this project.
     - This is an individual project. As stated in Section 6.6, you must identify the source of the code that you have used, whether it comes from COMP9334 or public domain.
     - Do not request help from anyone other than the teaching staff of COMP9344.
     - Do not post your project work or code to the course forum.
     - project submissions are routinely examined both automatically and manually for work written by others.

   Rationale: this project is designed to develop the individual skills needed to solve problems. Using work/code written by, or taken from, other people will stop you learning these skills. Other CSE courses focus on skills needed for working in a team.

- The use of AI generative tools, such as ChatGPT, is not permitted on this project.

  Rationale: We have given you the permission to use public domain code as a basis to develop your project, so it is not necessary for you to use ChatGPT. Our test with ChatGPT found that it was not able to supply us with a piece of complete running code for simulating a M/M/1 queue.
- Sharing, publishing, or distributing your project work is not permitted.
  - Do not provide or show your project work to any other person, other than the teaching staff of COMP9334. For example, do not message your work to friends.
  - Do not publish your project code via the Internet. For example, do not place your project in a public GitHub repository.

  Rationale: by publishing or sharing your work, you are facilitating other students using your work. If other students find your project work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.
- Sharing, publishing, or distributing your project work after the completion of COMP9334 is not permitted.
  - For example, do not place your project in a public GitHub repository after this offering of COMP9334 is over.

  Rationale: COMP9334 may reuse project themes covering similar concepts and content. If students in future terms find your project work and submit part or all of it as their own work, you may become involved in an academic integrity investigation.

References

[1] Mor Harchol-Balter. Performance Modeling and Design of Computer Systems. Cambridge University Press (2013).

[2] Benjamin Berg et al., The case for phase-aware scheduling of parallelizable jobs. *Performance Evaluation*, Volume 153, February 2022. `https://doi.org/10.1016/j.peva.2021.102246`