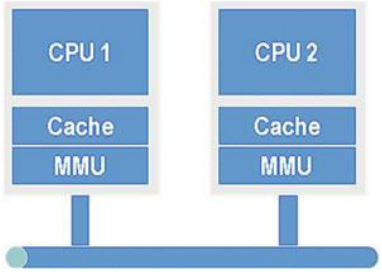
Multiprocessor Scheduling

Our discussion on processor scheduling so far has focused on the scheduling of a single processor from a single ready queue. Today, we are going to look at some scheduling issues in multiprocessor systems and other types of processing environments. We may need to continue the very end of the planned material at the start of tomorrow's class.

Improving System Throughput



Adding more processors to a system seems the obvious way to increase overall system throughput. The scheduling optimization problem becomes more complex and again there is no single best solution which ensures optimal usage of the processors for all workload cases.

Contention for the bus, I/O, shared memory, operating system code and maintaining cache coherency can cause processor stalling, so doubling the number of processors does not usually double the performance or throughput of the system if they share those resources. If they have access to independent resources then there still needs to be communication of workload data and control amongst the processing elements which impacts performance.

Amdahl's law states that the speedup achieved by using N processors instead of 1 is found from the following formula:-

$$\text{Speedup}(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Serial part of job = 1 (100%) - Parallel part

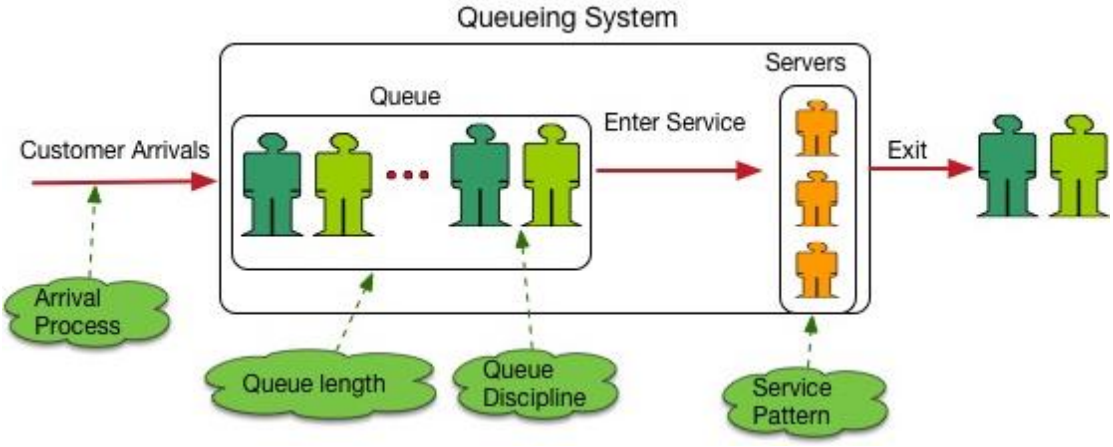
Parallel part is divided up by N workers

P is the percentage of the work that can be done in parallel. So for example, if 50% of the work can be done in parallel, then the speedup resulting from using 2 processors instead of 1 would be 1/((1-0.5)+0.5/2) = 1/0.75 = 4/3, so just 1.33 of an increase not 2. If you had 4 processors instead of 1, the time for the serial part is the same (1 - 0.5) and the time for the parallel part is a little quicker at 0.5/4 so speedup overall is 8/5 = 1.6, not 4. With 100 processors the speedup is only 1.98, not 100.
So parallel processing is best on tasks where only a tiny fraction of the work is serial.

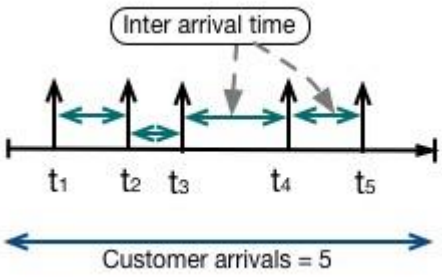
Increasing throughput effectively is a common aim of many services, not just computer systems and is the focus of queuing theory applicable in many service management situations, not just computing.

Queuing Theory

In modeling and managing the performance of any system, we must describe the arrival pattern of tasks, how tasks are queued and the speed of the service.

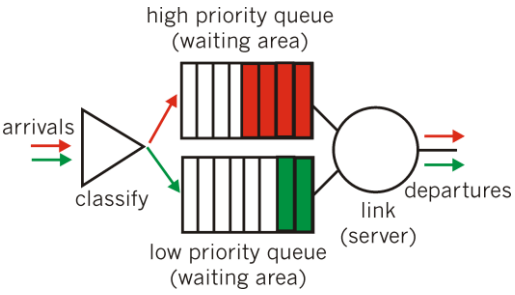


Every real world system might have its own unique arrival characteristics, physical queue configurations and service capacity. Queuing theory, modelling and simulation has the purpose of assisting managers of those services in how best to organise the entire system in order to reduce the delays and improve user satisfaction with faster service at the least cost.



Arrival and Queuing Process

Some example patterns, in using a service customers might arrive by Appointment to a consultant (known predictable workload), In groups to a restaurant (bursts of tasks arriving together), At a particular time of night to a taxi service (predictable heavy load times) or just Randomly



Queuing and Dispatch Policy

There might be multiple queues, a VIP or FASTPASS priority queue or public/private queue or other type classification for example.

There may be a policy of customers being turned away if the service decides it is too busy or has no space to accommodate the queue and so on.

Arrival Pattern for a Computer System

In a computer system, let's say the arrival pattern of tasks may be **random**, with **single independent tasks** arriving **one at a time** over the interval. Many tasks might arrive close together at some times, few at other times and maybe sometimes periods where none at all arrive but assume the arrival of tasks is completely random. For modelling purposes, we let λ represent the average rate of tasks arriving into such a system during an observation period.

What are the chances of more tasks in that period?

We might like to design system to cope with fluctuations. One way to estimate the arrival distribution might be to monitor the system and note the time between task arrivals and then using statistical techniques, fit a probability formula to the data so that we can estimate the number of arrivals in any given period.

Probability of events for a Poisson distribution

An event can occur 0, 1, 2, ... times in an interval.

The average number of events in an interval is designated λ (lambda).

The probability of observing k events in an interval is given by the equation

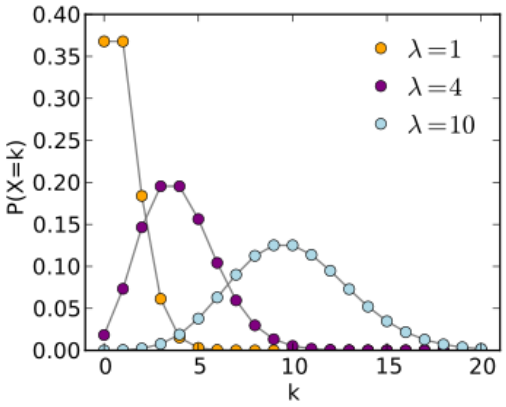
$$P(k \text{ events in interval}) = \frac{\lambda^k e^{-\lambda}}{k!}$$

where

- λ is the average number of events per interval
- e is the number 2.71828... (Euler's number) the base of the natural logarithms
- k takes values 0, 1, 2, ...
- $k! = k \times (k-1) \times (k-2) \times \ldots \times 2 \times 1$ is the factorial of k.

This equation is the probability mass function (PMF) for a Poisson distribution.

A Poisson Distribution, for example, can be used with the arrival process we described to our computer system in order to find the **probability** of a specific number of tasks arriving within a given time period.



The Poisson distribution for different average arrival rates λ are shown. If the average task arrival rate λ is 1 (orange dots) for example, there is about a 37% probability that 0 and a 37% probability that 1 task will arrive in the given period and virtually 0% probability that 5 tasks will arrive. So this type of statistical formula gives us an idea of the service performance we might like to set up to deal with the incoming load. We might put in place a service that can process 2 jobs per time unit and this would deal with over 90% of likely arrival rates where λ=1. Putting in a service that could deal with 5 jobs per unit of time might be a waste of resources.

Queuing Process & Service System

We assume tasks will remain in the system until completed. In real world queuing systems, the queuing pattern might be more dynamic, where sometimes arriving customers look at the length of a queue and balk at joining it, or sometimes they join for a while and then leave it due to excessive waiting time. Sometimes customers might jockey between one queue and another hoping to get serviced more quickly and so on, but here we ignore those effects on system performance and presume all our tasks arrive randomly, are patient in the queue and well behaved!

The queueing system may have a particular configuration such as being composed of a single queue or multiple queues. It might have physical configuration limits, a finite number of places, and it may be serviced according to a particular discipline, for example First Come First Server, Random, or based on Priority. Tasks are submitted from the queue to the service system when resources are available based on queue service policy.

Arrivals ➔ Queue ➔ Being Served ➔ Done

λ (Lambda) – Average arrival rate
μ (Mu) – Average service rate

1/λ = Average time between arrivals
1/μ = Average time for service

The speed of a service system can be expressed in two ways:- The service rate μ describes the average number of tasks serviced during a particular time period, and the average service time indicates the amount of time needed to service a task. One is the reciprocal of the other.

Service times may be static (like a machine filling a coffee cup) or follow a more variable distribution pattern that can be modelled by a formula, such as Exponential Distribution with a calculable average processing time per task.

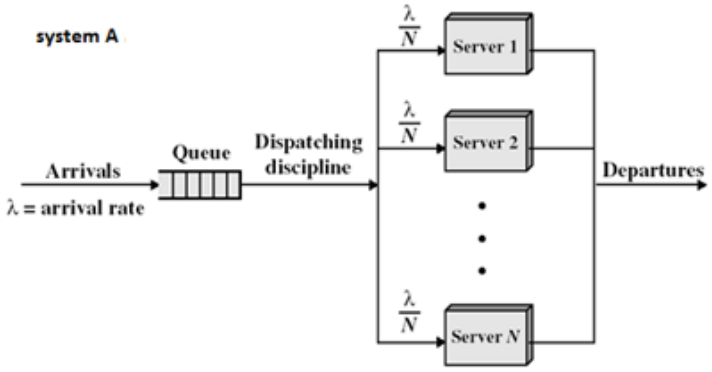
So if a processor can service 5 tasks in 1 second, the service rate is 5 tasks per second and the service time is 0.2 seconds per task on average. If the average arrival rate is 2 tasks per second then the processor will be 40% busy on average.

$$p = \frac{\lambda}{\mu} = \text{average system utilization}$$

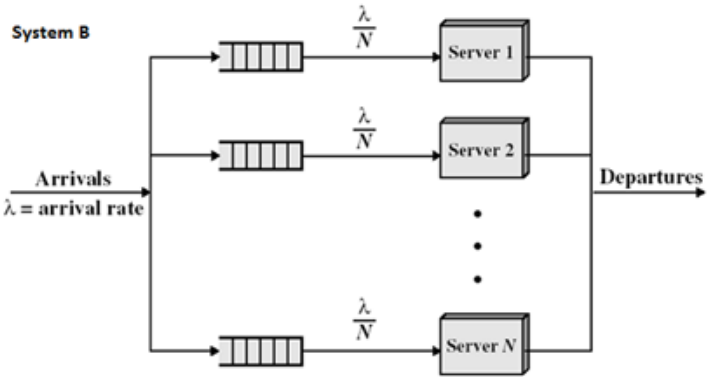
Note : $\mu > \lambda$ for system stability. If this is not the case, an infinitly long line will eventually form.

Multiprocessor System Organisation

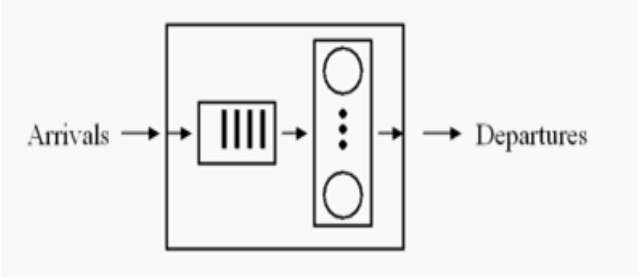
A multiprocessor system can be composed of a ***heterogeneous*** (different types) or ***homogenous*** (all the same) collection of processors.



With homogeneous processors, a serpentine queue, or single queue of tasks may be serviced by all processors A homogeneous system can use asymmetric multiprocessing or symmetric multiprocessing (SMP). The serpentine queue in a bank, ticket counter or airline check-in desk serviced by a group of tellers is an example of this model using symmetric multiprocessing where each teller independently calls the next person (usually) from the top of the queue.



With heterogeneous systems, assuming the processors have different machine architectures and instruction sets, each processor type must have its own separate queue of processes, whose binary images can be interpreted by that processor type. The multiple queue model is generally used in supermarkets for queuing at a checkout. Remember there are physical parameters that affect the choice of queuing model in a supermarket such as organising the space for the queue(s) and the travel time to push a trolley from the head of a queue to a checkout. These physical parameters do not occur with tasks queuing on a processor system.



If the average arrival rate of tasks in a Poisson system is λ and service pattern follows an exponential distribution with an average service rate μ, then it can be proven (from Little's Law) that the mean residence or turnaround time in the system (time spent in the box, departure time – arrival time) *T*, is related to λ and μ by the formula:-

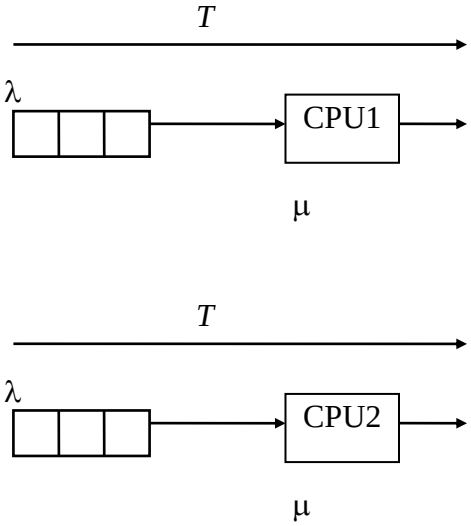
$$T = \frac{1}{\mu - \lambda}$$

Of course μ must be greater than λ for the system to be able to cope with the load.

For example, a system receives five requests per second and has capacity to process 10 requests per second (average of 0.1 sec service time per request). The average time T a task would expect to spend in the system would be 0.2 seconds.

The Supermarket Model

If we take n single processor systems, each with their own queue then we get:-



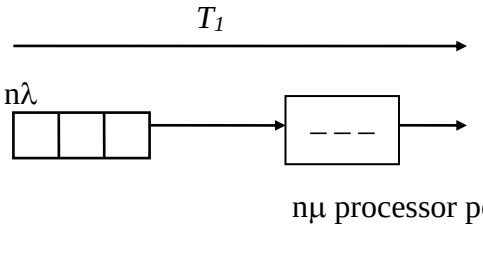
where

$$T = \frac{1}{\mu - \lambda}$$

for each processor. No matter which system a process joins, the expected turnaround time is the same if the systems have the same arrival rates and service rates.

The Bank Model

Instead if we combine the queues into a single queue serviced by n processors we get:-



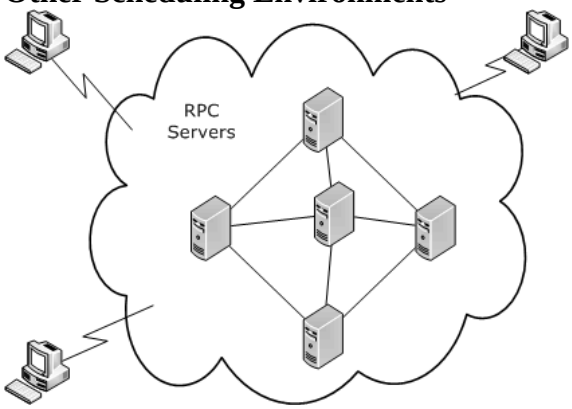
where

$$T_1 = \frac{1}{n\mu - n\lambda} = \frac{T}{n}$$ i.e a Faster Mean Turnaround Time

Assuming of course that all processors can be made equally busy all of the time and the arrival rate is random.

Although it is more daunting for a customer to join a single long queue, you can see that they should be in it for less time on average with the same number of servers.

Other Scheduling Environments



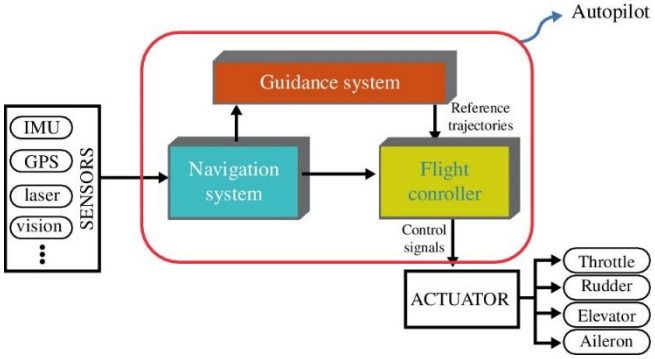
A **Distributed System** is composed of a collection of physically distributed computer systems connected by a local area network. From a scheduling perspective, a distributed operating system would endeavour to monitor and **balance the load** on each of these autonomous systems in addition to sharing other network resources in a seamless and transparent fashion. The term load balancing implies an effort to distribute load evenly among processing nodes usually requiring a relatively expensive deterministic analysis. The term load sharing is a less computationally intensive, more heuristic approach, which attempts to identify lightly loaded or heavily loaded nodes and either offload or give them a share of some work but with suboptimal workload assignment among nodes.

Scheduling in distributed systems is even more complex than in centralised multiprocessor systems, because the system information is generally not accessible in one single place but is managed on separate nodes in the network. Algorithms must gather information needed for decision making and transmit control information using message exchanges across the network. Message communication can be subject to delays, corruption and loss all leading to greater complexity within distributed algorithms.

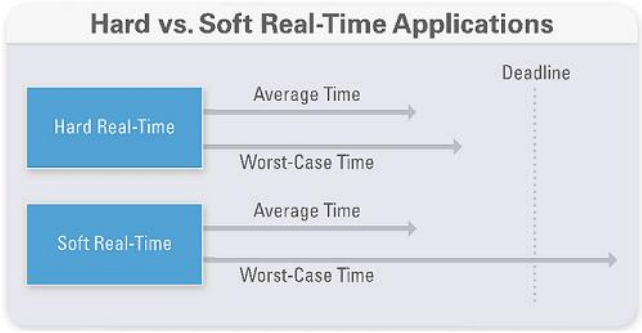
Real Time Systems

Some computer systems are designed for specific process control applications. For example, consider a production line where raw materials are supplied at one end of the line and finished goods come out the other.

Along the production line, sensors, valves and other actuators are monitored and controlled by the computer system. Sensors bring data to the computer which must be analysed and subsequently results in modifications to valves and actuators.



A real time system has well defined, fixed time constraints. Processing must be done within those constraints or the system may fail.

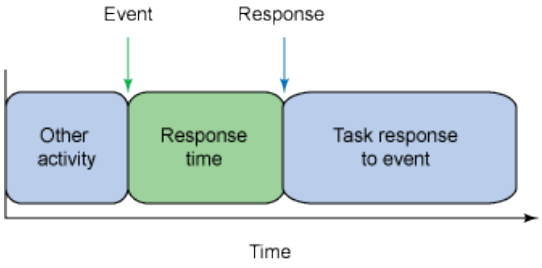


Hard Real Time Systems are required to complete a critical task within a guaranteed amount of time. When a process is submitted to the system, it is accompanied by a statement of the amount of time within which it needs to complete processing. The real time scheduler either admits the process, guaranteeing that the process will complete on time, or it rejects the request.

Note: Such guarantees are impossible in systems which use virtual memory due to unpredictable delays in servicing page faults. Hard real time systems therefore use specialised operating systems whose process behaviour and function timing is predictable. Such systems might have limited general purpose use.

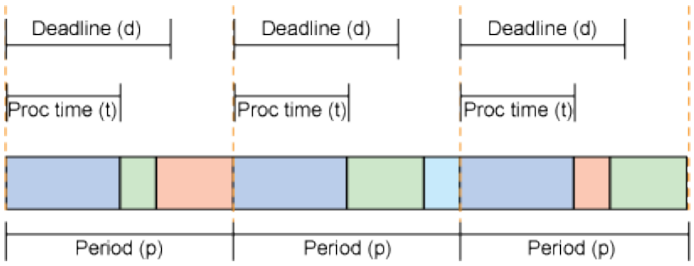
Soft Real Time Systems are ones which endeavour to meet scheduling deadlines but where missing an occasional deadline may be tolerable. Critical processes are typically given **higher priority** than others and their priority does not degrade over time.

Dispatch Latency



One of the key things when priority based approaches are used it that the **dispatch latency** must be small to ensure that a real time process can start executing quickly. This requires that system calls be preemptible. In other words, when the operating system is working on something for another process, it must be possible to preempt the operating system itself if a higher priority real time process want to run.

Examples of Real Time Scheduling Algorithms



Earliest Deadline First
When an event is detected, the handling process is added to the ready queue. The list is kept sorted by deadline, which for a periodic event is the next time of occurrence of the event. The scheduler services processes from the front of the sorted queue.

Least Laxity Algorithm
If a process requires 200msec and must finish in 250msec, then its laxity is 50msec. The Least Laxity Algorithm chooses the process with the smallest amount of time to spare. This algorithm might work better in situations where events occur at irregular intervals.