Queuing Theory without probabilities
Queueing Theory without probabilities

- **Queueing system**
  - servers + waiting rooms
  - customers arrive, wait, get served, depart or go to next server
  - queueing disciplines
    - non-preemptive: fifo, priority, ...
    - preemptive: round-robin, multi-level feedback, ...

- Operating systems are examples of queueing systems
  - servers: hw/sw resources (cpu, disk, req handler, …)
  - customers: PCBs, TCBs, ...

- Given: arrival rates, service times, queueing disciplines, ...
- Obtain: queue sizes, response times, fairness, bottlenecks, ...

---

**Queueing Theory without probabilities**

- Consider cars traveling on a road with a turn
  - each car takes 3 seconds to go through the turn
  - at most one car can be in the turn at any time
  - \( N(t) \): # cars in the turn and waiting to enter the turn

- Load < 1: stable w/ waits depending on burstiness
- Load > 1: unstable, ever-increasing waits

---

![Diagram of car arrivals and turns](image)
Queuing Theory without probabilities

- Assume unending stream of customers:
  - arrival rate $\lambda$ or $X$: \# arrivals per second
  - average service time $S$: work needed per customer
  - average response time $R$: departure time $D$ - arrival time $A$
  - average wait time $W$: response time - service time
  - throughput $X$: \# departures per sec averaged over all time
  - average customers in system $N$: waiting or busy
  - utilization $U$: fraction of time server is busy

- Typical goal
  - Given: arrival rate, avg service time, queueing discipline
  - Obtain: average response time, average queue size

- Little’s Law (for any steady-state system):
  - $N = \lambda \times R$

---

Queuing Theory without probabilities

- Avg queue size $N$ increases exponentially with load $\rho$
  - becoming $\infty$ as $\rho \to 1$
  - $N$ increases as burstiness increases
**FCFS non-preemptive**

<table>
<thead>
<tr>
<th>customer</th>
<th>( A_i )</th>
<th>( S_i )</th>
<th>( D_i )</th>
<th>( R_i )</th>
<th>( W_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>4.0</td>
<td>3.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>6.0</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>7.0</td>
<td>4.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

repeats every 10 seconds

- System becomes empty at time 7 —> stable
  - Average response time:
    \[
    R = \frac{3.0 + 4.0 + 4.0}{3} = \frac{11.5}{3} \text{ sec}
    \]
  - Average wait time:
    \[
    W = \frac{0.0 + 2.0 + 3.5}{3} = \frac{5.5}{3} \text{ sec}
    \]
  - Arrival rate = throughput:
    \[
    \lambda = \frac{3}{10} \text{ arrivals / sec}
    \]
  - Utilization:
    \[
    U = \frac{6}{10}
    \]
  - Average number customers:
    \[
    N = \lambda \times R = \frac{3}{10} \times \frac{11.5}{3} = \frac{11.5}{10}
    \]

**SJF non-preemptive**

<table>
<thead>
<tr>
<th>customer</th>
<th>( A_i )</th>
<th>( S_i )</th>
<th>( D_i )</th>
<th>( R_i )</th>
<th>( W_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>4.0</td>
<td>3.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>7.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>5.0</td>
<td>2.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

repeats every 10 seconds

- System becomes empty at time 7 —> stable
  - Average response time:
    \[
    R = \frac{3.0 + 5.0 + 2.5}{3} = \frac{10.5}{3} \text{ sec}
    \]
  - Average wait time:
    \[
    W = \frac{0.0 + 3.0 + 1.5}{3} = \frac{4.5}{3} \text{ sec}
    \]
  - Arrival rate = throughput:
    \[
    \lambda = \frac{3}{10} \text{ arrivals/sec}
    \]
  - Utilization:
    \[
    U = \frac{6}{10}
    \]
  - Average number customers:
    \[
    N = \lambda \times R = \frac{3}{10} \times \frac{10.5}{3} = \frac{10.5}{10}
    \]
SJS preemptive

<table>
<thead>
<tr>
<th>customer</th>
<th>$A_i$</th>
<th>$S_i$</th>
<th>$D_i$</th>
<th>$R_i$</th>
<th>$W_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>5.0</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>7.0</td>
<td>5.0</td>
<td>3.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>3.5</td>
<td>1.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

repeats every 10 seconds

- System becomes empty at time 7 $\rightarrow$ stable
  - Average response time: $R = \frac{4.0 + 5.0 + 1.0}{3} = \frac{10.0}{3}$ sec
  - Average wait time: $W = \frac{1.0 + 3.0 + 0.0}{3} = \frac{4.0}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals / sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{10.0}{3} = \frac{10}{10}$

RR preemptive

<table>
<thead>
<tr>
<th>customer</th>
<th>$A_i$</th>
<th>$S_i$</th>
<th>$D_i$</th>
<th>$R_i$</th>
<th>$W_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3.0</td>
<td>7.0</td>
<td>6.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>2.0</td>
<td>6.0</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>2.5</td>
<td>1.0</td>
<td>5.0</td>
<td>2.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>

repeats every 10 seconds

- System becomes empty at time 7 $\rightarrow$ stable
  - Average response time: $R = \frac{6.0 + 4.0 + 2.5}{3} = \frac{12.5}{3}$ sec
  - Average wait time: $W = \frac{3.0 + 2.0 + 1.5}{3} = \frac{6.5}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals / sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{12.5}{3} = \frac{12.5}{10}$
Scheduling Summary

- response time vs average num customers:

<table>
<thead>
<tr>
<th></th>
<th>FCFS</th>
<th>SJF</th>
<th>SJFP</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>11.5</td>
<td>11.5</td>
<td>10.5</td>
<td>10.5</td>
</tr>
<tr>
<td>$N$</td>
<td>11.5</td>
<td>11.5</td>
<td>10.5</td>
<td>10.5</td>
</tr>
</tbody>
</table>

Little's Law: $N = \lambda \times R$

- ratio of $R/N$ constant, because throughput ($X$) is same

M/M/1 queues

- M/M/1 assumption
  - (M)emoryless (independent) interarrival times exp dist w/ mean $\frac{1}{\lambda}$
  - (M)emoryless (independent) service times exp dist w/ mean $S$
  - (1) server

- For stable M/M/1 queue: $N = \frac{\rho}{1 - \rho}$
Deadlocks

- Necessary conditions for deadlock
  - Mutual exclusion - Threads claim exclusive control of resources
  - Hold and wait - Threads hold resources while waiting for additional resources
  - No preemption - Resources cannot be removed from threads that hold them
  - Circular wait - There exists a chain of threads such that each holds one or more resources that are requested by the next thread in the chain

- What to do?
  - prevent
  - avoid
  - deal with when they occur
  - pretend they never happen

Resource Allocation Graph

A set of vertices \( V \) and a set of edges \( E \):

- \( V \) is partitioned into two types:
  - \( P = \{P_1, P_2, \ldots, P_n\} \), the set of all the processes in the system
  - \( R = \{R_1, R_2, \ldots, R_m\} \), the set of all resource types in the system

- request edge: directed edge \( P_i \rightarrow R_j \)
- assignment edge: directed edge \( R_j \rightarrow P_i \)
Resource Allocation Graph (cont.)

- Process

- Resource type with 4 instances

- $P_i$ requests instance of $R_j$

- $P_i$ is holding an instance of $R_j$

Resource Allocation Graph example

- $P_1$ requesting instance of $R_1$
- $P_2$ requesting instance of $R_2$
- one $R_1$ held by $P_1$
- one $R_2$ held by $P_3$
- distinct $R_3$ instances held by $P_1$ and $P_2$
Resource Allocation Graph 

- $P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_2$  
  deadlock
- $R_3 \rightarrow P_1 \rightarrow R_1 \rightarrow P_2$  
  not deadlock, but blocked by deadlock

Handling Deadlocks what to do

- What to do?
  - prevent
  - avoid
  - deal with when they occur
  - pretend they never happen
Deadlock Prevention

- Try to prevent one of the four conditions from holding true
  - Difficult to eliminate mutual exclusion
  - Prevent threads from requesting new resources when holding other resources (eliminates hold and wait)
  - Require threads not immediately able to get all needed resources to give up those they have (eliminates no preemption)
  - Require agreed-upon resource acquisition ordering (eliminates circular waiting).

Agree on lexicographic ordering on lock acquisitions:

- or address-based:

```c
if (m1 > m2) {
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```
Deadlock Prevention circular wait

- Agree on lexicographic ordering on lock acquisitions:

  T1: `pthread_mutex_lock(m1);`  
  `pthread_mutex_lock(m2);`
  T2: `pthread_mutex_lock(m2);`  
  `pthread_mutex_lock(m1);`

- or address-based:

  ```c
  if (m1 > m2) {        // grab in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
  } else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
  }
  ```
Deadlock Prevention **hold and wait**

- Acquire all locks at once:
  
  ```c
  pthread_mutex_lock(prevention);       // begin acquisition
  pthread_mutex_lock(L1);
  pthread_mutex_lock(L2);
  ...
  pthread_mutex_unlock(prevention);    // end
  ```

- **But:**
  - `prevention` lock is global
  - need complete information

Deadlock Prevention **no preemption**

- **Try locks**
  - atomically grab lock if available, or return w/ error
  
  ```c
  top:   
  pthread_mutex_lock(L1);               // begin acquisition
  if (pthread_mutex_trylock(L2) != 0) {
      pthread_mutex_unlock(L1);
      goto top;
  }
  ```

- Works even if other thread choose different order. However: **livelock:**
  - Possible, though unlikely, that the threads both repeatedly back off. We could fix this with random delays.

- Other issue is **encapsulation:** some of the locks might be acquired in called functions, making jump back to initial state more difficult
Deadlock Prevention mutual exclusion

- **Lock-free** and **wait-free** data structures and algorithms
- use atomic instructions such as *CompareAndSwap*

// pseudocode of atomic assembly instruction

```c
int CompareAndSwap(int *address, int expected, int new) {
    if (*address == expected) {
        *address = new; // success
        return 1;
    }
    return 0; // failure
}
```

- Use with the following:

```c
void AtomicIncrement(int *value, int amount) {
    do {
        int old = *value;
    } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

Deadlock Prevention more wait-free

// mutex-based

```c
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    n->value = value;
    pthread_mutex_lock(listlock); // begin critical section
    n->next  = head;
    head     = n;
    pthread_mutex_unlock(listlock); // end critical section
}
```

// fixed

```c
void insert(int value) {
    node_t *n = malloc(sizeof(node_t)); assert(n != NULL);
    n->value = value;
    do{
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```
Deadlock Avoidance  

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

- System is in *safe state* if there exists:
  - sequence \(<P_1, P_2, \ldots, P_n>\) of ALL the processes in the systems such that for each \(P_i\), the resources that \(P_i\) can still request can be satisfied by currently available resources + resources held by all \(P_j\) s.t. \(j < i\)

- That is:
  - If \(P_i\)'s resource needs are not immediately available, then \(P_i\) can wait until all \(P_j\) have finished
  - When \(P_j\) is finished, \(P_i\) can obtain needed resources, execute, return allocated resources, and terminate
  - When \(P_i\) terminates, \(P_{i+1}\) can obtain its needed resources, …

Deadlock Avoidance  

- In other words:
  - System is in safe state \(\rightarrow\) no deadlocks
  - System is in unsafe state \(\rightarrow\) possibility of deadlocks
  - Avoidance of unsafe states ensure no deadlocks.
Deadlock Avoidance 

safe states

- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of resource types
  - Use the banker’s algorithm

Deadlock Avoidance 

safe states

- New \textit{claim} edge $P_i \rightarrow R_j$ indicates $P_i$ may request resource $R_j$. (represented by dashed line)
- \textit{Claim} edge converts to \textit{request} edge when a process requests a resource
- \textit{Request} edge converted to an \textit{assignment} edge when the resource is allocated to the process
- When a resource is released by a process, \textit{assignment} edge reconverts to a \textit{claim} edge

- Resources must be \textit{claimed} a priori in the system.
Deadlock Avoidance safe states

A request by $P_i$ for resource $R_j$ can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.

Deadlock Mitigation dealing with it

- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if $P_i$ is waiting for resource held by $P_j$

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.

- An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph.
Deadlock Mitigation dealing with it

- Construct the waits for graph
- Check for cycles
- Pick *any* thread of the cycle and kill it

Deadlock Mitigation ignoring it

“Not everything worth doing is worth doing well” - Tom West

- Consequence may be
  - minor
  - very rare