Memory Hierarchies

- disk
- RAM
- Cache
- Registers

Faster

Larger, Slower
Optimal Caching

- We want to store $n > k$ items in a cache $C$ of size $k$
- We’re given a request order of items: 1, 5, 8, 9, . . .
- There’s a hit if $i$ is in $C$ when it is requested, miss otherwise
- At each request, we can swap a cache item with a non-cache item
- **Our Goal:** Minimize the number of misses
Example

Suppose $k = 3$, with $n = 4$ items \{a, b, c, d\}.

Request order: a, b, c, a, d, d, c, d, a, b

<table>
<thead>
<tr>
<th>Request:</th>
<th>Cache:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(start)</td>
<td>abc</td>
</tr>
<tr>
<td>a</td>
<td>abc</td>
</tr>
<tr>
<td>d</td>
<td>abd</td>
</tr>
<tr>
<td>d</td>
<td>abd</td>
</tr>
<tr>
<td>c</td>
<td>acd</td>
</tr>
<tr>
<td>d</td>
<td>acd</td>
</tr>
<tr>
<td>a</td>
<td>acd</td>
</tr>
<tr>
<td>b</td>
<td>abd</td>
</tr>
</tbody>
</table>
Belady’s Algorithm:

- When $d_i$ is requested but not in the cache, evict the cache item that will next be used farthest into the future.

How can we prove this results in the fewest number of misses possible?
Notes

- Normally, in practice, we don’t know the full sequence of requests.

- So we can’t calculate which item will be used farthest in the future.

- Heuristics such as least recently used are employed instead.

- Still important to know the OPT, however: we can compare to OPT to figure out how well our heuristic did.
Same basic approach as before: we transform some optimal schedule into our schedule without increasing the number of misses.

- Let $D$ be a sequence of requests.
- Let $A$ be the schedule for $D$ obtained by the “farthest in the future” algorithm.
- Let $S_j$ be a schedule for $D$ that makes the same decisions as $A$ up through the first $j$ steps.

- We’ll construct a schedule $S_{j+1}$ that agrees with $A$ up through the first $j + 1$ steps and has the same number of misses as $S_j$. 
The Exchange Argument

- Start with a schedule $S_0$.
- Let $A$ be the schedule produced by our algorithm.
- We’ll start with some optimal solution $S_0$.
- We make local changes to $S_0$, trying to transform it into $A$.
- Each local change will keep the same number of misses.
If we let the first schedule $S_0 = OPT$, then this shows that $A$ has the same number of misses as OPT.
We’ll construct a schedule $S_{j+1}$ that agrees with $A$ up through the first $j + 1$ steps and has the same number of misses as $S_j$. 

Given:

\begin{align*}
A &\to e \\
S_j &\to f \\
&\to g \\
&\to f \\
&\to e \\
&\to h \\
&\to \text{Case (i)} \\
&\to \text{Case (ii)}
\end{align*}

Want to construct:

\begin{align*}
S_{j+1} &\to e \\
&\to g \\
&\to f \\
&\to e \\
&\to h
\end{align*}
After step $j$, the cache contents of $S_j$ and $S_{j+1}$ differ slightly:

$$S_j : \quad - \quad - \quad - \quad - \quad e$$

$$S_{j+1} : \quad - \quad - \quad - \quad - \quad f$$

When one of the following happens, we can make their cache contents agree:

1. **$S_j$ evicts $e$ because some $g \not\in \{f, e\}$ was requested**: We evict $f$ and now $S_j$ and $S_{j+1}$ have the same cache content.

2. **$S_j$ evicts some element $h$ because $f$ was requested**: $f$ is already in $S_{j+1}$’s cache, so we use the chance to bring $e$ in.
S_{j+1} will have the same \# of misses as S_j unless item e is accessed before we have a chance to make the caches the same.

But this can’t happen, because e was the item that is accessed farthest in the future (by our greedy rule).

Hence, before we access e next, we’ll have a chance to fix the cache up.
Greedy Scheduling Recap:

- **Interval Scheduling**: *Increasing Finishing Time.*
  Greedy stays ahead of OPT, can always do what OPT does.

- **Interval Partitioning**: *Increasing Start Time.*
  Limit on how much of the resource (colors) have been used.

- **Minimizing Lateness**: *Increasing Deadlines.*
  Violations of this rule can only increase the cost.
  (oversimplified)

- **Optimal Caching**: *Farthest in the Future.*