CSC 252: Computer Organization
Fall 2021: Lecture 18

Wrap-up caching
Processes

Instructor: Alan Beadle

Department of Computer Science
University of Rochester
Announcements

Midterm solutions online

You can take your exam back once you have checked for grading errors

For integrity reasons, please bring up possible grading mistakes before you leave with your exam; Not after.

A4 is online; I will send an announcement tonight
General Cache Organization (S, E, B)

- \( E = 2^e \) lines per set
- \( S = 2^s \) sets
General Cache Organization \((S, E, B)\)

\[ E = 2^e \text{ lines per set} \]

\[ S = 2^s \text{ sets} \]

\[ B = 2^b \text{ bytes per cache line (the data)} \]
General Cache Organization (S, E, B)

- \( E = 2^e \) lines per set
- \( S = 2^s \) sets
- \( B = 2^b \) bytes per cache line (the data)
- valid bit
- set
- line
General Cache Organization (S, E, B)

E = $2^e$ lines per set

S = $2^s$ sets

B = $2^b$ bytes per cache line (the data)

dirty bit (if write-back)

valid bit
General Cache Organization (S, E, B)

E = $2^e$ lines per set

S = $2^s$ sets

Cache size:
$C = S \times E \times B$ data bytes

Overhead:
Tag, valid bit, dirty bit. Plus bits for implementing replacement policy (not shown).

B = $2^b$ bytes per cache line (the data)
Cache Access

\[ E = 2^e \text{ lines per set} \]

\[ S = 2^s \text{ sets} \]

Address of word:

- **t** bits
- **s** bits
- **b** bits

- tag
- set
- index
- offset
Cache Access

E = 2^e lines per set

S = 2^s sets

Address of word:
- t bits
- s bits
- b bits

• Locate set

- tag
- set index
- Line offset
Cache Access

- **S** = $2^s$ sets
- **E** = $2^e$ lines per set

Address of word:
- **t** bits
- **s** bits
- **b** bits

- **t bits**
- **s bits**
- **b bits**

- **tag**
- **set**
- **Line index offset**

- **Locate set**
- **Check if any line in set has matching tag**
- **Yes + line valid: hit**

**valid bit**  **B = $2^b$ bytes per cache line (the data)**
Cache Access

\[ E = 2^e \text{ lines per set} \]

\[ S = 2^s \text{ sets} \]

\[ B = 2^b \text{ bytes per cache line (the data)} \]

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

Address of word:
- \( t \) bits
- \( s \) bits
- \( b \) bits

- tag
- set
- Line index offset

data begins at this offset

valid bit

0 1 2 \( \cdots \) B-1

v tag
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes

S = 2^s sets

Address of char:

\[ \text{t bits} \begin{array}{c}
\text{0...01} \\
\text{100}
\end{array} \]
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes

$S = 2^s$ sets

Address of char:

$\begin{array}{c|c}
\text{t bits} & 0...01 & 100 \\
\end{array}$

find set
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes

Address of char: t bits 0...01 100
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes

(valid? +match: assume yes = hit)

Address of \texttt{char}:
\begin{tabular}{c|c|c}
\hline
\texttt{t bits} & 0...01 & 100 \\
\hline
\end{tabular}

Byte 4 is here
Example: Direct Mapped Cache

Direct mapped: One line per set
Assume: cache line size 8 bytes

If tag doesn’t match: old line is evicted and replaced
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):
0 [0000₂],
1 [0001₂],
7 [0111₂],
8 [1000₂],
0 [0000₂]

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>0</td>
<td>?</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0 \[0000_2]\), miss
1 \[0001_2]\),
7 \[0111_2]\),
8 \[1000_2]\),
0 \[0000_2]\)

\[\begin{array}{cccc}
\text{Set 0} & \text{Set 1} & \text{Set 2} & \text{Set 3} \\
0 & ? & ? & ? \\
\end{array}\]
Direct-Mapped Cache Simulation

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Set 0</th>
<th>1</th>
<th>0</th>
<th>M[0-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 $\text{[0000_2]}$, miss
1 $\text{[0001_2]}$
7 $\text{[0111_2]}$
8 $\text{[1000_2]}$
0 $\text{[0000_2]}$
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0 \[0000_2\], miss
1 \[0001_2\],
7 \[0111_2\],
8 \[1000_2\],
0 \[0000_2\]

The two bytes at memory address 0 and 1
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

```
0  [0000_2],  miss
1  [0001_2],  hit
7  [0111_2],
8  [1000_2],
0  [0000_2]
```

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Set 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The two bytes at memory address 0 and 1
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):
0 \[0000_2\], miss
1 \[0001_2\], hit
7 \[0111_2\], miss
8 \[1000_2\],
0 \[0000_2\]

The two bytes at memory address 0 and 1
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0  [00002], miss
1  [00012], hit
7  [01112], miss
8  [10002],
0  [00002]

The two bytes at memory address 0 and 1
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address Index</th>
<th>Memory Location</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000_2]</td>
<td>0</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>[0001_2]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[0111_2]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000_2]</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000_2]</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

The two bytes at memory address 0 and 1
The two bytes at memory address 6 and 7
Direct-Mapped Cache Simulation

- 4-bit address space, i.e., Memory = 16 bytes
- \( B = 2 \) bytes/line, \( S = 4 \) sets, \( E = 1 \) line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[00002],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[00012],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[01112],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[10002],</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[00002]</td>
<td></td>
</tr>
</tbody>
</table>

The two bytes at memory address 0 and 1
The two bytes at memory address 6 and 7
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Line 1</th>
<th>Line 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
<td></td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
<td></td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
<td></td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The two bytes at memory address 8 and 9

The two bytes at memory address 6 and 7
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂]</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂]</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂]</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td>miss</td>
</tr>
</tbody>
</table>

The two bytes at memory address 8 and 9
The two bytes at memory address 6 and 7

v | Tag | Line
---|-----|------
Set 0 | 1 | 1 | M[8-9]
Set 1 | | | 
Set 2 | | | 
Set 3 | 1 | 0 | M[6-7]
Direct-Mapped Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>t=1</th>
<th>s=2</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>xx</td>
<td>x</td>
</tr>
</tbody>
</table>

- 0 [0000<sub>2</sub>], miss
- 1 [0001<sub>2</sub>], hit
- 7 [0111<sub>2</sub>], miss
- 8 [1000<sub>2</sub>], miss
- 0 [0000<sub>2</sub>], miss

The two bytes at memory address 0 and 1

The two bytes at memory address 6 and 7
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache line size 8 bytes

Address of short int:

\[
\begin{array}{c}
\text{t bits} \\
0...01 \quad 100
\end{array}
\]
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache line size 8 bytes

Address of `short int`:

```
| t bits | 0...01 | 100 |
```

```
v  tag  0 1 2 3 4 5 6 7
v  tag  0 1 2 3 4 5 6 7
v  tag  0 1 2 3 4 5 6 7
v  tag  0 1 2 3 4 5 6 7
v  tag  0 1 2 3 4 5 6 7
v  tag  0 1 2 3 4 5 6 7
v  tag  0 1 2 3 4 5 6 7
v  tag  0 1 2 3 4 5 6 7
```
E-way Set Associative Cache (Here: $E = 2$)

$E = 2$: Two lines per set
Assume: cache line size 8 bytes

Address of short int:

```
| t bits | 0...01 | 100 |
```
E-way Set Associative Cache (Here: $E = 2$)

$E = 2$: Two lines per set
Assume: cache line size 8 bytes

Address of short int: $t$ bits 0...01 100

compare both
E-way Set Associative Cache (Here: \( E = 2 \))

E = 2: Two lines per set
Assume: cache line size 8 bytes

Address of short int:

\[
\begin{array}{c}
t \text{ bits} \\
0...01 \\
100
\end{array}
\]

compare both

valid? + match: yes = hit
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache line size 8 bytes

Address of short int:

0...01 100

t bits

(valid? + match: yes = hit)

Offset within a line
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache line size 8 bytes

Address of short int:

Offset within a line

short int (2 Bytes) is here

compare both

valid? + match: yes = hit

v tag 0 1 2 3 4 5 6 7 v tag 0 1 2 3 4 5 6 7

t bits 0...01 100
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache line size 8 bytes

No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU),
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):
0 [0000₂],
1 [0001₂],
7 [0111₂],
8 [1000₂],
0 [0000₂]

Set 0 0 0 0
Set 1 0 0 0
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>t=2</th>
<th>s=1</th>
<th>b=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Tag</td>
<td>Line</td>
</tr>
<tr>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 0

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Set 1

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):
0  [0000_2], miss
1  [0001_2],
7  [0111_2],
8  [1000_2],
0  [0000_2]

Set 0
<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Set 1
<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

- t=2, s=1, b=1
- 4-bit address space, i.e., Memory = 16 bytes
- S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>v</strong></td>
<td><strong>v</strong></td>
</tr>
<tr>
<td><strong>Tag</strong></td>
<td><strong>Tag</strong></td>
</tr>
<tr>
<td><strong>Line</strong></td>
<td><strong>Line</strong></td>
</tr>
</tbody>
</table>

- 0: [0000₂], miss
- 1: [0001₂], hit
- 7: [0111₂],
- 8: [1000₂],
- 0: [0000₂]
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th></th>
<th>[0000₂]</th>
<th>[0001₂]</th>
<th>[0111₂]</th>
<th>[1000₂]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>miss</td>
<td>hit</td>
<td>miss</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 0  
- v: 1  
- Tag: 00  
- Line: M[0-1]  
- 0

Set 1  
- 0

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>Set 1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# 2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

### Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

### Set 0

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Set 1

<table>
<thead>
<tr>
<th>1</th>
<th>01</th>
<th>M[6-7]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>1</td>
<td>[0001₂],</td>
<td>hit</td>
</tr>
<tr>
<td>7</td>
<td>[0111₂],</td>
<td>miss</td>
</tr>
<tr>
<td>8</td>
<td>[1000₂],</td>
<td>miss</td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

Set 0

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 1

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):

<table>
<thead>
<tr>
<th>Address</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0000₂], miss</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[0001₂], hit</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>[0111₂], miss</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>[1000₂], miss</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>[0000₂]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>M[0-1]</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>M[8-9]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set 1</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>M[6-7]</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-Way Set Associative Cache Simulation

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 cache lines/set

Address trace (reads, one byte per read):

0  [0000_2], miss
1  [0001_2], hit
7  [0111_2], miss
8  [1000_2], miss
0  [0000_2]  hit

<table>
<thead>
<tr>
<th>v</th>
<th>Tag</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Set 1</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>

|      |      |       |
| 0    |      |       |

xx 00 00 00 00 2
xx 01 01 01 01 2
xx 10 10 10 10 2
Today

• Processes and Signals: running multiple programs concurrently
  • Processes
    • Process Control
  • Signals
Processes

• Definition: A *process* is an instance of a running program.
  • One of the most profound ideas in computer science
  • Not the same as “program” or “processor”
Processes

• Definition: A process is an instance of a running program.
  • One of the most profound ideas in computer science
  • Not the same as “program” or “processor”

• Process provides each program with two key abstractions:
Processes

• Definition: A *process* is an instance of a running program.
  • One of the most profound ideas in computer science
  • Not the same as “program” or “processor”

• Process provides each program with two key abstractions:
  • “Owns” the CPU
Processes

• Definition: A process is an instance of a running program.
  • One of the most profound ideas in computer science
  • Not the same as “program” or “processor”

• Process provides each program with two key abstractions:
  • “Owns” the CPU
    • Each program seems to have exclusive use of the CPU
Processes

• Definition: A *process* is an instance of a running program.
  • One of the most profound ideas in computer science
  • Not the same as “program” or “processor”

• Process provides each program with two key abstractions:
  • “Owns” the CPU
    • Each program seems to have exclusive use of the CPU
    • Done by the OS kernel through “context switching”
Processes

• Definition: A process is an instance of a running program.
  • One of the most profound ideas in computer science
  • Not the same as “program” or “processor”

• Process provides each program with two key abstractions:
  • “Owns” the CPU
    • Each program seems to have exclusive use of the CPU
    • Done by the OS kernel through “context switching”
  • Private address space
Processes

• Definition: A process is an instance of a running program.
  • One of the most profound ideas in computer science
  • Not the same as “program” or “processor”

• Process provides each program with two key abstractions:
  • “Owns” the CPU
    • Each program seems to have exclusive use of the CPU
    • Done by the OS kernel through “context switching”
  • Private address space
Processes

• Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

• Process provides each program with two key abstractions:
  - *“Owns” the CPU*
    - Each program seems to have exclusive use of the CPU
    - Done by the OS kernel through “context switching”
  - *Private address space*
    - Each program seems to have exclusive use of main memory.
Processes

- Definition: A process is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - “Owns” the CPU
    - Each program seems to have exclusive use of the CPU
    - Done by the OS kernel through “context switching”
  - Private address space
    - Each program seems to have exclusive use of main memory.
    - Provided by OS through “virtual memory”
Multiprocessing: The Illusion

- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices
### Multiprocessing Example

- Running program “top” on Unit/Linux
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

---

**Partial Output of Running “top”**

```
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkededit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

<table>
<thead>
<tr>
<th>PID</th>
<th>COMMAND</th>
<th>%CPU</th>
<th>TIME</th>
<th>#TH</th>
<th>#WQ</th>
<th>#PORT</th>
<th>#MREG</th>
<th>RPRVT</th>
<th>RSHRD</th>
<th>RSIZE</th>
<th>VPRVT</th>
<th>VSIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>99217</td>
<td>Microsoft Of</td>
<td>0.0</td>
<td>02:28.34</td>
<td>4</td>
<td>1</td>
<td>202</td>
<td>418</td>
<td>21M</td>
<td>24M</td>
<td>21M</td>
<td>66M</td>
<td>763M</td>
</tr>
<tr>
<td>99051</td>
<td>usbmuxd</td>
<td>0.0</td>
<td>00:04.10</td>
<td>3</td>
<td>1</td>
<td>47</td>
<td>66</td>
<td>436K</td>
<td>216K</td>
<td>480K</td>
<td>60M</td>
<td>2422K</td>
</tr>
<tr>
<td>99006</td>
<td>iTunesHelper</td>
<td>0.0</td>
<td>00:01.23</td>
<td>2</td>
<td>1</td>
<td>55</td>
<td>78</td>
<td>728K</td>
<td>312K</td>
<td>112K</td>
<td>43M</td>
<td>2429M</td>
</tr>
<tr>
<td>84286</td>
<td>bash</td>
<td>0.0</td>
<td>00:00.11</td>
<td>1</td>
<td>0</td>
<td>20</td>
<td>24</td>
<td>224K</td>
<td>732K</td>
<td>484K</td>
<td>17M</td>
<td>2378M</td>
</tr>
<tr>
<td>84285</td>
<td>xterm</td>
<td>0.0</td>
<td>00:00.83</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>73</td>
<td>656K</td>
<td>872K</td>
<td>692K</td>
<td>9728K</td>
<td>2382M</td>
</tr>
<tr>
<td>55939</td>
<td>Microsoft Ex</td>
<td>0.3</td>
<td>21:58.97</td>
<td>10</td>
<td>3</td>
<td>360</td>
<td>954</td>
<td>16M</td>
<td>65M</td>
<td>46M</td>
<td>114M</td>
<td>1057M</td>
</tr>
<tr>
<td>54751</td>
<td>sleep</td>
<td>0.0</td>
<td>00:00.00</td>
<td>1</td>
<td>0</td>
<td>17</td>
<td>20</td>
<td>92K</td>
<td>212K</td>
<td>360K</td>
<td>9632K</td>
<td>2370M</td>
</tr>
<tr>
<td>54739</td>
<td>launchdadd</td>
<td>0.0</td>
<td>00:00.00</td>
<td>2</td>
<td>1</td>
<td>33</td>
<td>50</td>
<td>488K</td>
<td>220K</td>
<td>1736K</td>
<td>48M</td>
<td>2409M</td>
</tr>
<tr>
<td>54737</td>
<td>top</td>
<td>6.5</td>
<td>00:02.53</td>
<td>1/1</td>
<td>0</td>
<td>30</td>
<td>29</td>
<td>1416K</td>
<td>216K</td>
<td>2124K</td>
<td>17M</td>
<td>2378M</td>
</tr>
<tr>
<td>54719</td>
<td>automountd</td>
<td>0.0</td>
<td>00:00.02</td>
<td>7</td>
<td>1</td>
<td>53</td>
<td>64</td>
<td>860K</td>
<td>216K</td>
<td>2184K</td>
<td>53M</td>
<td>2413M</td>
</tr>
<tr>
<td>54701</td>
<td>ocspd</td>
<td>0.0</td>
<td>00:00.05</td>
<td>4</td>
<td>1</td>
<td>61</td>
<td>54</td>
<td>1268K</td>
<td>2644K</td>
<td>3132K</td>
<td>50M</td>
<td>2426M</td>
</tr>
<tr>
<td>54661</td>
<td>Grab</td>
<td>0.6</td>
<td>00:02.75</td>
<td>6</td>
<td>3</td>
<td>222+</td>
<td>389+</td>
<td>15M+</td>
<td>26M+</td>
<td>40M+</td>
<td>75M+</td>
<td>2556M+</td>
</tr>
<tr>
<td>54659</td>
<td>cooked</td>
<td>0.0</td>
<td>00:00.15</td>
<td>2</td>
<td>1</td>
<td>40</td>
<td>61</td>
<td>3316K</td>
<td>224K</td>
<td>4088K</td>
<td>42M</td>
<td>2411M</td>
</tr>
</tbody>
</table>
```

---

Note: The output is truncated for readability. The full output can be found in the reference material.
Multiprocessing Illustration

Memory

Process 1
- Stack
- Heap
- Data
- Code

Process 2
- Stack
- Heap
- Data
- Code

Process N
- Stack
- Heap
- Data
- Code

CPU
- Registers
Multiprocessing Illustration

Memory

Process 1
- Stack
- Heap
- Data
- Code

Process 2
- Stack
- Heap
- Data
- Code

Process N
- Stack
- Heap
- Data
- Code

CPU
- Registers
Multiprocessing Illustration

![Multiprocessing Illustration Diagram]

- **Process 1**: Stack, Heap, Data, Code
- **Saved registers**
- **CPU Registers**
- **Process 2**: Stack, Heap, Data, Code
- **Process N**: Stack, Heap, Data, Code

Memory
Multiprocessing Illustration

Memory

Process 1
- Stack
- Heap
- Data
- Code
- Saved registers

Process 2
- Stack
- Heap
- Data
- Code

Process N
- Stack
- Heap
- Data
- Code

CPU
- Registers
Multiprocessing Illustration

- Process 1
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- Process 2
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- Process N
  - Stack
  - Heap
  - Data
  - Code
Multiprocessing Illustration

Process 1
- Stack
- Heap
- Data
- Code
- Saved registers

Process 2
- Stack
- Heap
- Data
- Code
- Saved registers

Process N
- Stack
- Heap
- Data
- Code

CPU
- Registers
Multiprocessing Illustration

Memory

Process 1
- Stack
- Heap
- Data
- Code
- Saved registers

Process 2
- Stack
- Heap
- Data
- Code
- Saved registers

Process N
- Stack
- Heap
- Data
- Code
- Saved registers

CPU
- Registers
Multiprocessing Illustration

- **Process 1**
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- **Process 2**
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- **Process N**
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers
Multiprocessing Illustration

- **Process 1**
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- **Process 2**
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- **Process N**
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- **Memory**
  - CPU
  - Registers
Multiprocessing Illustration

Context switch managed by the OS. Not controllable by programmers.
Multiprocessing: The Multi-Core Case

- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
Concurrent Processes

• Each process is a logical control flow.
• Two processes run concurrently (are concurrent) if their flows overlap in time
• Otherwise, they are sequential
Concurrent Processes

• Each process is a logical control flow.
• Two processes run concurrently (are concurrent) if their flows overlap in time
• Otherwise, they are sequential
• Examples (running on single core):

```
Time

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Concurrent Processes

• Each process is a logical control flow.
• Two processes run concurrently (are concurrent) if their flows overlap in time
• Otherwise, they are sequential
• Examples (running on single core):
  • Concurrent: A & B, A & C
Concurrent Processes

• Each process is a logical control flow.
• Two processes run concurrently (are concurrent) if their flows overlap in time.
• Otherwise, they are sequential.
• Examples (running on single core):
  • Concurrent: A & B, A & C
  • Sequential: B & C
User View of Concurrent Processes

• Control flows for concurrent processes are physically disjoint in time

• However, we can think of concurrent processes as running in parallel with each other
Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the **kernel**
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.

- Control flow passes from one process to another via a **context switch**

![Diagram of context switching between two processes](image-url)
Today

• Processes and Signals: running multiple programs concurrently
  • Processes
  • Process Control
  • Signals
Obtaining Process IDs

• `pid_t getpid(void)`
  • Returns PID of current process

• `pid_t getppid(void)`
  • Returns PID of parent process
Creating and Terminating Processes

From a programmer’s perspective, we can think of a process as being in one of three states

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel

- **Stopped**
  - Process execution is suspended and will not be scheduled until further notice (through something call *signals*)

- **Terminated**
  - Process is stopped permanently
Terminating Processes

• Process becomes terminated for one of three reasons:
  • Receiving a signal whose default action is to terminate
  • Returning from the main routine
  • Calling the exit function

• `void exit(int status)`
  • Terminates with an exit status of status
  • Convention: normal return status is 0, nonzero on error
  • Another way to explicitly set the exit status is to return an integer value from the main routine

• `exit` is called once but never returns.
Creating Processes

- Parent process creates a new running child process by calling `fork`

- `int fork(void)`
  - Returns 0 to the child process, child’s PID to parent process
  - Child is almost identical to parent:
    - Child get an identical (but separate) copy of the parent’s (virtual) address space (i.e., same stack copies, code, etc.)
    - Child gets identical copies of the parent’s open file descriptors
    - Child has a different PID than the parent

- `fork` is interesting (and often confusing) because it is called once but returns twice
fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}

fork.c
```

```
linux> ./fork
parent: x=0
child : x=2
```
fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

- Call once, return twice
**fork Example**

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child

```bash
linux> ./fork
parent: x=0
child : x=2
```
fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child
- Duplicate but separate address space
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent

```
linux> ./fork
parent: x=0
cmp child: x=2
```
fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child
- Duplicate but separate address space
  - x has a value of 1 when fork returns in parent and child
  - Subsequent changes to x are independent
- Shared open files
  - stdout is the same in both parent and child

```
linux> ./fork
parent: x=0
child: x=2
```
Process Address Space

- Kernel space
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by `malloc`)
- Read/write data segment (`.data, .bss`)
- Read-only code segment (`.init, .text, .rodata`)
- Unused

Memory invisible to user code

- `%rsp` (stack pointer)
- `brk`

Loaded from the executable file

Program Counter

0x400000

0
Process Address Space

- Kernel space
- User stack (created at runtime)
- Memory-mapped region for shared libraries
- Run-time heap (created by malloc)
- Read/write data segment (.data, .bss)
- Read-only code segment (.init, .text, .rodata)
- Unused

Memory invisible to user code

- %rsp (stack pointer)
- brk

Loaded from the executable file

Program Counter
What Happens at `fork()`?

Code Segment

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        x++;  // 2
        exit(0);
    }

    /* Parent */
    x--;  // 0
    exit(0);
}
```
What Happens at `fork()`?

Parent Address Space

Stack

\[ x = 1 \]

Code Segment

```c
int main()
{
  pid_t pid;
  int x = 1;

  pid = Fork();
  if (pid == 0) {
    /* Child */
    x++; // 2
    exit(0);
  }

  /* Parent */
  x--; // 0
  exit(0);
}
```
What Happens at `fork()`?

Parent Address Space

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    pid_t pid;
    int x = 1;
    pid = Fork();
    if (pid == 0) {
        /* Child */
        x++; // 2
        exit(0);
    }
    /* Parent */
    x--; // 0
    exit(0);
}
```
What Happens at `fork()`?

**Parent** Address Space

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        x++; // 2
        exit(0);
    }

    /* Parent */
    x--; // 0
    exit(0);
}
```

**Child** Address Space

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        x++; // 2
        exit(0);
    }

    /* Parent */
    x--; // 0
    exit(0);
}```
What Happens at `fork()`?

**Parent** Address Space

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        x++; // 2
        exit(0);
    }

    /* Parent */
    x--; // 0
    exit(0);
}
```

**Child** Address Space

```
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {
        /* Child */
        x++; // 2
        exit(0);
    }

    /* Parent */
    x--; // 0
    exit(0);
}```
Creating Processes

• Parent process creates a new child process by calling `fork`
• Child get an identical (but separate) copy of the parent’s (virtual) address space (i.e., same stack copies, code, etc.)
• `int fork(void)`
  • Returns 0 to the child process
  • Returns `child’s PID` to the parent process
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}

fork.c
Interpreting Process Graphs

• Original graph:

• Abstracted graph:
Interpreting Process Graphs

• Original graph:

• Abstracted graph:

Feasible execution ordering:
Interpreting Process Graphs

• Original graph:

• Abstracted graph:

Feasible execution ordering:

Infeasible execution ordering: