Direct-Mapped Example

• Suppose our addresses are 16 bits long.

• Our cache has 16 entries, block size of 16 bytes
  – 4 bits in address for the index
  – 4 bits in address for byte offset
  – Remaining bits (8): tag
Let’s say we access memory at address: 0110101100110100

Step 1:
- Partition address into tag, index, offset

<table>
<thead>
<tr>
<th>Line</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (16 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Direct-Mapped Example

• Let’s say we access memory at address:
  – 01101011 0011 0100

• Step 1:
  – Partition address into tag, index, offset
Direct-Mapped Example

• Let’s say we access memory at address:
  – 01101011 0011 0100

• Step 2:
  – Use index to find line (row)
  – 0011 -> 3
Direct-Mapped Example

• Let’s say we access memory at address:
  – 01101011 0011 0100

• Step 2:
  – Use index to find line (row)
  – 0011 -> 3
Direct-Mapped Example

Let’s say we access memory at address:
- 01101011 0011 0100

Step 3:
- Check the tag
- Is it 01101011 (hit)?
- Something else (miss)?
- (Must also ensure valid)
How would the cache change if we performed the following memory operations?

- Read 01000100 (Value: 5)
- Read 11100010 (Value: 17)
- Write 01110000 (Value: 7)
- Read 10101010 (Value: 12)
- Write 01101100 (Value: 2)

1. Write dirty line to memory.
2. Load new value, set it to 2, mark it dirty (write).
Associativity

• Problem: suppose we’re only using a small amount of data (e.g., 8 bytes, 4-byte block size)

• Bad luck: (both) blocks map to same cache line
  – Constantly evicting one another
  – Rest of cache is going unused!

•Associativity: allow a set blocks to be stored at the same index. Goal: reduce conflict misses.
A. **In exactly one place. (“Direct-mapped”)**
   - Every location in memory is directly mapped to one place in the cache. Easy to find data.

B. **In a few places. (“Set associative”)**
   - A memory location can be mapped to (2, 4, 8) locations in the cache. Middle ground.

C. In most places, but not all.

D. **Anywhere in the cache. (“Fully associative”)**
   - No restrictions on where memory can be placed in the cache. Fewer conflict misses, more searching.
Comparison

Direct-mapped
• Tag tells you if you found the correct data.
• Offset specifies which byte within block.
• Middle bits (index) tell you which 1 line to check.
• (+) Low complexity, fast.
• (-) Conflict misses.

N-way set associative
• Tag tells you if you found the correct data.
• Offset specifies which byte within block.
• Middle bits (set) tell you which N lines to check.
• (+) Fewer conflict misses.
• (-) More complex, slower, consumes more power.
Comparison: 1024 Lines
(For the same cache size, in bytes.)

**Direct-mapped**
- 1024 indices (10 bits)

**2-way set associative**
- 512 sets (9 bits)
  - Tag slightly (1 bit) larger.

<table>
<thead>
<tr>
<th>Set #</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>508</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>509</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>510</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>511</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2-Way Set Associative

<table>
<thead>
<tr>
<th>Tag (20 bits)</th>
<th>Set (9 bits)</th>
<th>Byte offset (3 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3941</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

#### Table:

<table>
<thead>
<tr>
<th>Set #</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4063</td>
<td></td>
<td>1</td>
<td>0</td>
<td>3941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>509</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>510</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>511</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Same capacity as previous example: 1024 rows with 1 entry vs. 512 rows with 2 entries.
2-Way Set Associative

<table>
<thead>
<tr>
<th>Tag (20 bits)</th>
<th>Set (9 bits)</th>
<th>Byte offset (3 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3941</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>508</td>
</tr>
<tr>
<td>509</td>
</tr>
<tr>
<td>510</td>
</tr>
<tr>
<td>511</td>
</tr>
</tbody>
</table>

Check all locations in the set, in parallel.
2-Way Set Associative

<table>
<thead>
<tr>
<th>Tag (20 bits)</th>
<th>Set (9 bits)</th>
<th>Byte offset (3 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3941</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set #</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4063</td>
<td></td>
<td>1</td>
<td>0</td>
<td>3941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>509</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>510</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>511</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multiplexer

Select correct value.
4-Way Set Associative Cache

Clearly, more complexity here!
Eviction

• Mechanism is the same...
  – Overwrite bits in cache line: update tag, valid, data

• Policy: choose which line in the set to evict
  – Pick a random line in set
  – Choose an invalid line first
  – Choose the least recently used block
    • Has exhibited the least locality, kick it out!

Common combo in practice.
Least Recently Used (LRU)

- Intuition: if it hasn’t been used in a while, we have no reason to believe it will be used soon.

- Need extra state to keep track of LRU info.

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4063</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (8 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3941</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Least Recently Used (LRU)

• Intuition: if it hasn’t been used in a while, we have no reason to believe it will be used soon.

• Need extra state to keep track of LRU info.

Another reason why associativity often maxes out at 8 or 16.

These are metadata bits, not “useful” program data storage.

(Approximations make it not quite as bad.)
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>17</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>010</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>111 4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

We have a match. LRU bit stays the same
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.
Valid bit is 0, evict the left.

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>111</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>001</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>010</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 111000010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
<th>Valid bit</th>
<th>Eviction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0-1</td>
<td>0</td>
<td>111</td>
<td>4 17</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>9</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Valid bit is 0, evict the left.
LRU stays the same
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 011000100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used.
1 means the right line was used least recently.

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0-1</td>
<td>0</td>
<td>111</td>
<td>4 17</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How would the cache change if we performed the following memory operations? (2-way set)

Read `01000100` (Value: 5)
Read `11100010` (Value: 17)
Write `01100100` (Value: 7)
Read `01000110` (Value: 5)
Write `01100000` (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

Tag does not match and we have a miss! Valid bit is set for both. Which do we evict?

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0-1</td>
<td>0</td>
<td>111</td>
<td>4 17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>111</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

Tag does not match and we have a miss! LRU is the one on the left. More expected locality on the right. But it has a dirty bit set.

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0-1</td>
<td>0</td>
<td>111</td>
<td>4 17</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1-1</td>
<td>111</td>
<td>9</td>
<td></td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 011000100 (Value: 7)
Read 01000110 (Value: 5)
Write 011000000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

Are we done?
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 011000100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>111</td>
<td>4 17</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>1</td>
<td>10</td>
<td>111011</td>
<td>9 7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Set the LRU bit!
How would the cache change if we performed the following memory operations? (2-way set)

- Read 01000100 (Value: 5)
- Read 11100010 (Value: 17)
- Write 011000100 (Value: 7)
- Read 01000110 (Value: 5)
- Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used.
1 means the right line was used least recently.

Set the LRU bit!

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>111</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>111</td>
<td>011</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

We have a hit on the right. Update LRU bit. Avoids conflict misses from prev. instruction that would have occurred in a direct mapped cache!
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

<table>
<thead>
<tr>
<th>Set #</th>
<th>LRU</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Data (4 Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>111</td>
<td>4 17</td>
<td>1</td>
<td>0</td>
<td>001</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>01</td>
<td>1</td>
<td>111</td>
<td>9 7</td>
<td>1</td>
<td>0</td>
<td>010</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td>1</td>
<td>0</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

LRU of 0 means the left line in the set was least recently used. 1 means the right line was used least recently.

Tag is incorrect on both sides. Valid bit set for both, but right has LRU set.
How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01100100 (Value: 7)
Read 01000110 (Value: 5)
Write 01100000 (Value: 2)

LRU of 0 means the left line in the set was least recently used.
1 means the right line was used least recently.

Update the right-hand side, set dirty bit, set LRU to zero.
Cache Conscious Programming

• Knowing about caching and designing code around it can significantly effect performance

(ex) 2D array accesses

```
for(i=0; i < N; i++) {
    for(j=0; j< M; j++) {
        sum += arr[i][j];
    }
}
```

```
for(j=0; j < M; j++) {
    for(i=0; i < N; i++) {
        sum += arr[i][j];
    }
}
```

Algorithmically, both $O(N \times M)$.

Is one faster than the other?
Cache Conscious Programming

• Knowing about caching and designing code around it can significantly affect performance

(ex) 2D array accesses

Algorithmically, both $O(N \times M)$.

Is one faster than the other?

A. is faster.

B. is faster.

C. Both would exhibit roughly equal performance.
Cache Conscious Programming

The first nested loop is more efficient if the cache block size is larger than a single array bucket (for arrays of basic C types, it will be).

for (i=0; i < N; i++) {
    for (j=0; j < M; j++) {
        sum += arr[i][j];
    }
}

for (j=0; j < M; j++) {
    for (i=0; i < N; i++) {
        sum += arr[i][j];
    }
}

(ex) 1 miss every 4 buckets vs. 1 miss every bucket
Program Efficiency and Memory

• Be aware of how your program accesses data
  – Sequentially, in strides of size X, randomly, ...
  – How data is laid out in memory

• Will allow you to structure your code to run much more efficiently based on how it accesses its data

• Don’t go nuts...
  – Optimize the most important parts, ignore the rest
  – “Premature optimization is the root of all evil.” -Knuth
Amdahl’s Law

Idea: an optimization can improve total runtime at most by the fraction it contributes to total runtime

If program takes 100 secs to run, and you optimize a portion of the code that accounts for 2% of the runtime, the best your optimization can do is improve the runtime by 2 secs.

Amdahl’s Law tells us to focus our optimization efforts on the code that matters:

- Speed-up what is accounting for the largest portion of runtime to get the largest benefit. And, don’t waste time on the small stuff.
Up Next:

• Operating systems, Processes
Abstraction

- User / Programmer
  Wants low complexity

- Applications
  Specific functionality

- Software library
  Reusable functionality

- Operating system
  Manage resources

- Complex devices
  Compute & I/O
Abstraction

User / Programmer
Wants low complexity

Applications
Specific functionality

Software library
Reusable functionality

Operating system
Manage resources

Complex devices
Compute & I/O
OS Big Picture Goals

• OS is an extra code layer between user programs and hardware.

• Goal: Make life easier for users and programmers.

• How can the OS do that?
Key OS Responsibilities

1. Hardware gatekeeping and protection

2. Simplifying abstractions for programs (e.g., files)

3. Resource sharing (memory, CPU)
OS: Turn undesirable into desirable

• Turn undesirable inconveniences: reality
  – Complexity of hardware
  – Single processor
  – Limited memory
Before Operating Systems

• One program executed at a time...
Today: Multiprogramming

**Multiprogramming**: have multiple programs available to the machine, even if you only have one CPU core that can execute them.
Multiprogramming

Wait

Wait (for some resource)

Job 1 Running

Wait

Wait

Job 2 Running

Wait

Wait

Job 3 Running

Combined

CPU Time

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Lecture 6 – Slide 49
How many programs do you think are running on a typical desktop?

A. 1-10
B. 20-40
C. 40-80
D. 80-160
E. 160+
Running multiple programs

More than 200 processes running on a typical desktop!

• **Benefits:** when I/O issued, CPU not needed
  – Allow another program to run
  – Requires yielding and sharing memory

• **Challenges:** what if one running program...
  – Monopolizes CPU, memory?
  – Reads/writes another’s memory?
  – Uses I/O device being used by another?
OS: Turn undesirable into desirable

• Turn undesirable inconveniences: reality
  – Complexity of hardware
  – Single processor
  – Limited memory

• Into desirable conveniences: illusions
  – Simple, easy-to-use resources
  – Multiple/unlimited number of processors
  – Large/unlimited amount of memory
Virtualization

• Rather than exposing real hardware, introduce a “virtual”, abstract notion of the resource

• Multiple virtual processors
  – By rapidly switching CPU use

• Multiple virtual memories
  – By memory partitioning and re-addressing

• Virtualized devices
  – By simplifying interfaces, and using other resources to enhance function
Focus on the OS ‘kernel’

• “Operating system” has many interpretations
  – E.g., all software on machine minus applications (user
    or even limited to 3rd party)

• Our focus is the kernel
  – What’s necessary for everything else to work
  – Low-level resource control
  – Originally called the nucleus in the 60’s
The Kernel

• All programs depend on it
  – Loads and runs them
  – Exports system calls to programs

• Works closely with hardware
  – Accesses devices
  – Responds to interrupts

• Allocates basic resources
  – CPU time, memory space
  – Controls I/O devices: display, keyboard, disk, network
Kernel provides common functions

• Some functions useful to many programs
  – I/O device control
  – Memory allocation

• Place these functions in central place (kernel)
  – Called by programs (system calls)
  – Or accessed implicitly

• What should functions be?
  – How many programs should benefit?
  – Might kernel get too big?
• **Big Design Issue:** How do we make the OS efficient, reliable, and extensible?

• **General OS Philosophy:** The design and implementation of an OS involves a constant tradeoff between simplicity and performance.

• As a general rule, strive for simplicity.
  – except when you have a strong reason to believe that you need to make a particular component complicated to achieve acceptable performance
  – (strong reason = simulation or evaluation study)
Main Abstraction: The Process

• Abstraction of a running program
  – “a program in execution”
• Dynamic
  – Has state, changes over time
  – Whereas a program is static
• Basic operations
  – Start/end
  – Suspend/resume
Basic Resources for Processes

• To run, process needs some basic resources:
  – CPU
    • Processing cycles (time)
    • To execute instructions
  – Memory
    • Bytes or words (space)
    • To maintain state
  – Other resources (e.g., I/O)
    • Network, disk, terminal, printer, etc.
Machine State of a Process

- **CPU or processor context**
  - PC (program counter)
  - SP (stack pointer)
  - General purpose registers
- **Memory**
  - Code
  - Global Variables
  - Stack of activation records / frames
  - Other (registers, memory, kernel-related state)

Must keep track of these for every running process!
Resource Sharing

Reality
- Multiple processes
- Small number of CPUs
- Finite memory

Abstraction
- Process is all alone
- Process is always running
- Process has all the memory
Resource: CPU

• Many processes, limited number of CPUs.

• Each process needs to make progress over time. Insight: processes don’t know how quickly they should be making progress.

• Illusion: every process is making progress in parallel.
Timesharing: Sharing the CPUs

• Abstraction goal: make every process think it’s running on the CPU all the time.
  – Alternatively: If a process was removed from the CPU and then given it back, it shouldn’t be able to tell

• Reality: put a process on CPU, let it run for a short time (~10 ms), switch to another, ... (context switching)
How is Timesharing Implemented?

• Kernel keeps track of progress of each process
• Characterizes state of process’s progress
  – Running: actually making progress, using CPU
  – Ready: able to make progress, but not using CPU
  – Blocked: not able to make progress, can’t use CPU
• Kernel selects a ready process, lets it run
  – Eventually, the kernel gets back control
  – Selects another ready process to run, ...
Process State Diagram

- **State transitions**
  - Dispatch: allocate the CPU to a process
  - Preempt: take away CPU from process
  - Sleep: process gives up CPU to wait for event
  - Wakeup: event occurred, make process ready
## Kernel Maintains Process Table

<table>
<thead>
<tr>
<th>Process ID (PID)</th>
<th>State</th>
<th>Other info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1534</td>
<td>Ready</td>
<td>Saved context, ...</td>
</tr>
<tr>
<td>34</td>
<td>Running</td>
<td>Memory areas used, ...</td>
</tr>
<tr>
<td>487</td>
<td>Ready</td>
<td>Saved context, ...</td>
</tr>
<tr>
<td>9</td>
<td>Blocked</td>
<td>Condition to unblock, ...</td>
</tr>
</tbody>
</table>

- List of processes and their states
  - Also sometimes called “process control block (PCB)”
- Other state info includes
  - contents of CPU context
  - areas of memory being used
  - other information

Values of registers in use by process
Multiprogramming

• Given a running process
  – At some point, it needs a resource, e.g., I/O device
  – If resource is busy (or slow), process can’t proceed
  – “Voluntarily” gives up CPU to another process

• Mechanism: Context switching
Context Switching

• Allocating CPU from one process to another
  – First, save context of currently running process
  – Next, load context of next process to run
Context Switching

• Allocating CPU from one process to another
  – First, save context of currently running process
  – Next, load context of next process to run

• Loading the context
  – Load general registers, stack pointer, etc.
  – Load program counter (must be last instruction!)
How a Context Switch Occurs

- Process makes system call (TRAP) or is interrupted
  - These are the only ways of entering the kernel
- In hardware
  - Switch from user to kernel mode: amplifies power
  - Go to fixed kernel location: interrupt/syscall handler
- In software (in the kernel code)
  - Save context of last-running process
  - Conditionally
    - Select new process from those that are ready
    - Restore context of selected process
  - OS returns control to a process from interrupt/syscall
Why shouldn’t processes control context switching?

A. It would cause too much overhead.

B. They could refuse to give up the CPU.

C. They don’t have enough information about other processes.

D. Some other reason(s).
Time Sharing / Multiprogramming

• Given a running process
  – At some point, it needs a resource, e.g., I/O device
  – If resource is busy (or slow), process can’t proceed
  – “Voluntarily” gives up CPU to another process

• Mechanism: Context switching
• Policy: CPU scheduling
Managing Processes

• Given a process, how do we make it execute the program we want?

• Model: fork() a new process, execute program
Creating a Process

• One process can create other processes to do work.
  – The creator is called the parent and the new process is the child
  – The parent defines (or donates) resources and privileges to its children
  – A parent can either wait for the child to complete, or continue in parallel
fork()

- System call (function provided by OS kernel)

- Creates a duplicate of the requesting process
  - Process is cloning itself:
    - CPU context
    - Memory “address space”

(April) identical clones
The two processes are identical in every way, except for the return value of `fork()`.

- The child gets a return value of 0.
- The parent gets a return value of child’s PID.

```c
pid_t pid = fork(); // both continue after call
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Which process executes next? Child? Parent? Some other process?

Up to OS to decide. No guarantees. Don’t rely on particular behavior!
How many hello’s will be printed?

fork();
printf(“hello”);
if (fork()) {
    printf(“hello”);
}
fork();
printf(“hello”);

A. 6
B. 8
C. 12
D. 16
E. 18
How many hello’s will be printed?

fork();
printf(“hello”);
if (fork()) {
    printf(“hello”);
}
fork();
printf(“hello”);
Common `fork()` usage: Shell

• A “shell” is the program controlling your terminal (e.g., bash).

• It `fork()`’s to create new processes, but we don’t want a clone (another shell).

• We want the child to execute some other program: `exec()` family of functions.
exec ()

• Family of functions (execl, execlp, execv, ...).

• Replace the current process with a new one.

• Loads program from disk:
  – Old process is overwritten in memory.
  – Does not return unless error.
Common `fork()` usage: Shell

1. `fork()` child process.

2. `exec()` desired program to replace child’s address space.

3. `wait()` for child process to terminate.

The parent and child each do something different next.
Common `fork()` usage: Shell

1. `fork()` child process.
Common `fork()` usage: Shell

2. parent: `wait()` for child to finish

![Diagram](attachment:image.png)
Common `fork()` usage: Shell

2. child: `exec()` user-requested program
Common `fork()` usage: Shell

2. child: `exec()` user-requested program
Common `fork()` usage: Shell

3. child program terminates, cycle repeats
**Common `fork()` usage: Shell**

3. child program terminates, cycle repeats
Process Termination

• On process termination, the OS reclaims all resources assigned to the process.

• In Unix
  – a process can terminate itself using the `exit` system call.
  – a process can terminate a child using the `kill` system call.
Process Termination

• When does a process die?
  – It calls `exit(int status);`
  – It `returns (an int)` from `main`
  – It receives a termination signal (from the OS or another process)

• Key observation: the dying process *produces status information*.

• Who looks at this?
  • The parent process!
Reaping Children
(Bet you didn’t expect to see THAT title on a slide when you signed up for CS 31?)

- $\text{wait()}$: parents reap their dead children
  - Given info about why child died, exit status, etc.

- Two variants:
  - $\text{wait()}$: wait for and reap next child to exit
  - $\text{waitpid()}$: wait for and reap specific child

- This is how the shell determines whether or not the program you executed succeeded.
Common `fork()` usage: Shell

1. `fork()` child process.

2. `exec()` desired program to replace child’s address space.

3. `wait()` for child process to terminate.
   - Check child’s result, notify user of errors.

4. repeat...
What should happen if dead child processes are never reaped? (That is, the parent has not \texttt{wait()}ed on them?)

A. The OS should remove them from the process table (process control block / PCB).

B. The OS should leave them in the process table (process control block / PCB).

C. The neglected processes seek revenge as undead in the afterlife.
“Zombie” Processes

• Zombie: A process that has terminated but not been reaped by parent. (AKA defunct process)

• Does not respond to signals (can’t be killed)

• OS keeps their entry in process table:
  – Parent may still reap them, want to know status
  – Don’t want to re-use the process ID yet

Basically, they’re kept around for bookkeeping purposes, but that’s much less exciting...
Process Management: Summary

• A process is the unit of execution.
• Processes are represented as Process Control Blocks in the OS
  – PCBs contain process state, scheduling and memory management information, etc
• A process is either New, Ready, Waiting, Running, or Terminated.
• On a uniprocessor, there is at most one running process at a time.
• The program currently executing on the CPU is changed by performing a context switch
• Processes communicate either with message passing or shared memory
Signals

• How does a parent process know that a child has exited (and that it needs to call wait)?

• Signals: inter-process notification mechanism
  – Info that a process (or OS) can send to a process.
    • Please terminate yourself (SIGTERM)
    • Stop NOW (SIGKILL)
    • Your child has exited (SIGCHLD)
    • You’ve accessed an invalid memory address (SIGSEGV)
    • Many more (SIGWINCH, SIGUSR1, SIGPIPE, …)
Signal Handlers

• By default, processes react to signals according to the signal type:
  – SIGKILL, SIGSEGV, (others): process terminates
  – SIGCHLD, SIGUSR1: process ignores signal

• You can define “signal handler” functions that execute upon receiving a signal.
  – Drop what program was doing, execute handler, go back to what it was doing.
  – Example: got a SIGCHLD? Enter handler, call `wait()`
  – Example: got a SIGUSR1? Reopen log files.

• Some signals (e.g., SIGKILL) cannot be handled.
Summary

• Processes cycled off and on CPU rapidly
  – Mechanism: context switch
  – Policy: CPU scheduling

• Processes created by `fork()` ing

• Other functions to manage processes:
  – `exec()`: replace address space with new program
  – `exit()`: terminate process
  – `wait()`: reap child process, get status info

• Signals one mechanism to notify a process of something
Kernel Execution

• Great, the OS is going to somehow give us these nice abstractions.

• So...how / when should the kernel execute to make all this stuff happen?
Process vs. Kernel

• The kernel is the code that supports processes
  – System calls: fork ( ), exit ( ), read ( ), write ( ), ...
  – System management: context switching, scheduling, memory management
Kernel vs. Userspace: Model

Process 1
- OS
- Text
- Data
- Heap
- Stack

Process 2
- OS
- Text
- Data
- Heap
- Stack

Process N
- OS
- Text
- Data
- Heap
- Stack

System Calls:
- fork
- read
- write

Kernel:
- System Management
- Context Switching
- Scheduling
Kernel vs. Userspace: Model

Kernel

System Calls
fork
read
write

System Management
Context Switching
Scheduling

Process 1
OS
Text
Data
Heap
Stack

Process 2
OS
Text
Data
Heap
Stack

Process N
OS
Text
Data
Heap
Stack

Code:

Data:

Code + Data:
Kernel vs. Userspace: Model

Process 1
- OS
  - Text
  - Data
  - Heap
  - Stack

Process 2
- OS
  - Text
  - Data
  - Heap
  - Stack

Process N
- OS
  - Text
  - Data
  - Heap
  - Stack

Makes system call. OS accesses device, assigns resource, etc.
Kernel vs. Userspace: Model

OS has control. It will take care of process’s request, but it might take a while. It can context switch (and usually does at this point).
OS returns control to a process (not usually the same one).
Kernel vs. Userspace: Model

Transition is expensive, but often necessary.
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
Standard C Library Example

- C program invoking `printf()` library call, which calls `write()` system call

```c
#include <stdio.h>
int main ()
{
    ...
    ...
    printf("Greetings");
    ...
    return 0;
}
```
API – System Call – OS Relationship

user application

open ()

user mode

system call interface

kernel mode

Implementation of open ()
system call

return
Control over the CPU

- To context switch processes, kernel must get control:

1. Running process can give up control voluntarily
   - To block, call yield() to give up CPU
   - Process makes a blocking system call, e.g., read()
   - Control goes to kernel, which dispatches new process

2. CPU is forcibly taken away: preemption
CPU Preemption

1. While kernel is running, set a hardware timer.

2. When timer expires, a hardware interrupt is generated. (device asking for attention)

3. Interrupt pauses process on CPU, forces control to go to OS kernel.

4. OS is free to perform a context switch.