#include "csapp.h"

int main() {
    int *x = malloc(sizeof(int));
    *x = 10;
    if (Fork() == 0)
        printf("%d\n", *x);
    else {
        *x = 20;
        printf("%d\n", *x);
    }

    return 1;
}
#include "csapp.h"

int main() {
    int *x = malloc(sizeof(int));
    *x = 10;
    if (Fork() == 0)
        printf("%d\n", *x);
    else {
        *x = 20;
        printf("%d\n", *x);
    }

    return 1;
}
#include "csapp.h"

int main() {
    int *x = malloc(sizeof(int));
    *x = 10;
    if (Fork() == 0)
        printf("%p\n", x);
    else {
        *x = 20;
        printf("%p\n", x);
    }

    return 1;
}
#include "csapp.h"

int main() {
    int *x = malloc(sizeof(int));
    *x = 10;
    if (Fork() == 0)
        printf("%p\n", x);
    else {
        *x = 20;
        printf("%p\n", x);
    }

    return 1;
}
Physical vs. Virtual Addresses

CPU

Physical address 0x308

0x308
Physical vs. Virtual Addresses
Virtual Memory Benefits

✓ Isolates processes
Virtual Memory Benefits

✓ Isolates processes

✓ Simplifies memory management
  
  each process gets the uniform address space
Every Process’s View of Memory

x86_64 supports only 48-bit addresses, so kernel gets half of virtual space

0x7fffffffffffffff

... kernel virtual memory

stack
(created at runtime)

%rsp

shared libraries
(code and data)

brk

heap
(created by malloc)

read+write data
(.data, .bss)

loaded from executable

read-only data
(.text, .rodata)

unused

0x40000000

Virtual Memory Benefits

✓ Isolates processes

✓ Simplifies memory management
  each process gets the uniform address space

✓ Allows memory content to span devices
  ... especially main memory and disk

  virtual address range $\gg$ physical memory
Virtual Memory as a Cache

Page size typically 4k to 64k

“page” instead of “block”
“page fault” instead of “cache miss”

Fully associative

requires a large mapping

Complex replacement rules

instead of just LRU

Write-back

as opposed to write-through
Address Translation with a Page Table

Address size = $2^n$

Virtual address

$m$

Physical address

$\text{Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition}$
Address Translation with a Page Table

Address size = $2^n$  Page size = $2^p$

Virtual Page Number (VPN)  Virtual Page Offset (VPO)

$n-p$  $p$

Physical address

$m$
Address Translation with a Page Table

Address size = $2^n$  Page size = $2^p$

Virtual Page Number (VPN)  Virtual Page Offset (VPO)

$n-p$  $p$

Physical Page Number (PPN)  Physical Page Offset (PPO)

$m-p$  $p$
Address Translation with a Page Table

Address size = $2^n$  
Page size = $2^p$

Virtual Page Number (VPN)  
Virtual Page Offset (VPO)

$n-p$

page table

Valid  
Physical Page Number (PPN)

Physical Page Number (PPN)  
Physical Page Offset (PPO)

$m-p$  
$p$
Address Translation with a Page Table

Address size = $2^n$  Page size = $2^p$

page table base register (PTBR)

Virtual Page Number (VPN)  Virtual Page Offset (VPO)

\[ n-p \]

\[ p \]

page table

Valid  Physical Page Number (PPN)

Physical Page Number (PPN)  Physical Page Offset (PPO)

\[ m-p \]

\[ p \]
Address Translation with a Page Table

Specific to a process

Address size = $2^n$  Page size = $2^p$

Page table base register (PTBR)

Virtual Page Number (VPN)  Virtual Page Offset (VPO)

\[ n - p \]

\[ p \]

**Page table**

Valid  Physical Page Number (PPN)

Physical Page Number (PPN)  Physical Page Offset (PPO)

\[ m - p \]

\[ p \]

Address Translation with a Page Table

Address size = $2^n$  Page size = $2^p$

- Virtual Page Number (VPN) $n-p$
- Virtual Page Offset (VPO) $p$
- Physical Page Number (PPN) $m-p$
- Physical Page Offset (PPO) $p$

page table base register (PTBR)

Valid Physical Page Number (PPN)

not valid ⇒ page fault

Address Translation with a Page Table

Address size $= 2^n$  Page size $= 2^p$

page table base register (PTBR)

Virtual Page Number (VPN) Virtual Page Offset (VPO)

$\text{page table}$

Valid Physical Page Number (PPN)

Physical Page Number (PPN) Physical Page Offset (PPO)

$m-p$ $p$
Address Translation with a Page Table

Address size = $2^n$  Page size = $2^p$

page table base register (PTBR)

Virtual Page Number (VPN)  Virtual Page Offset (VPO)

$n-p$  $p$

page table

Valid  Physical Page Number (PPN)

Physical Page Number (PPN)  Physical Page Offset (PPO)

$m-p$  $p$

Table size = $2^{n-p}$

Multi-Level Page Table

- **page table base register (PTBR)**
- **Virtual Page Number (VPN)**
  - $VPN_1$, $VPN_2$, ..., $VPN_k$
- **Virtual Page Offset (VPO)**
- **Physical Page Number (PPN)**
- **Physical Page Offset (PPO)**

Diagram shows the mapping process from virtual to physical addresses through multiple levels of page tables.
Multi-Level Page Table

page table base register (PTBR)

higher levels allocated only as needed

VPN₁ | VPN₂ | ... | VPNₖ | Virtual Page Offset (VPO)

Physical Page Number (PPN) | Physical Page Offset (PPO)

Address Translation: Page Hit

Address Translation: Page Hit

CPU ➔ MMU

virtual address ➔ page table entry address

Address Translation: Page Hit

[Diagram showing the process of address translation with labels: CPU, MMU, virtual address, page table entry address]
Address Translation: Page Hit

CPU \rightarrow \text{virtual address} \rightarrow \text{MMU} \rightarrow \text{page table entry address} \rightarrow \text{physical address}
Address Translation: Page Hit

- CPU
  - virtual address
  - page table entry address
  - physical address
- MMU
- Page Table
Address Translation: Page Fault
Address Translation: Page Fault
Address Translation: Page Fault
Address Translation: Page Fault

- CPU
- MMU
- Disk
- page fault handler
- page table entry address
- virtual address

Address Translation: Page Fault

CPU → MMU
  → page table entry address
  → physical address
  → page fault handler
  → Disk

virtual address
Address Translation: Page Fault

- CPU
- MMU
- Physical address
- Page table entry address
- Page fault handler
- Disk

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Address Translation: Page Fault

Moving data to and from disk — ok if good \textit{locality}
Address Translation: Page Fault

Moving data to and from disk — ok if good *locality*

*Working set* = pages currently being used
Moving data to and from disk — ok if good **locality**

**Working set** = pages currently being used

Working set > physical memory ⇒ **thrashing**

Translation Lookaside Buffer

A *translation lookaside buffer* (TLB) is a custom cache for address translation.
A *translation lookaside buffer* (TLB) is a custom cache for address translation.
A translation lookaside buffer (TLB) is a custom cache for address translation.
Virtual Memory: User vs. Kernel Views

Disk

0xA4000
0xA5000
0xA6000
0xA7000

0xA60B5

0x1000
0x2000
0x3000

0xA60B5
Virtual Memory: User vs. Kernel Views
Virtual Memory: User vs. Kernel Views

```
0xA1000 0
0xA2000 0
0xA3000 0
0xA4000 1
0xA5000 1
0xA6000 1
0xA7000 0
0xA8000 0
0xA9000 0
0xAA000 0
```

...
Virtual Memory: User vs. Kernel Views

access unmapped page \(\Rightarrow\) segmentation fault

0xA1000 0
0xA2000 0
0xA3000 0
0xA4000 1
0xA5000 1
0xA6000 1
0xA7000 0
0xA8000 0
0xA9000 0
0xAA000 0

Disk

0xA60B5
0x1000
0x2000
0x3000

...
Virtual Memory: User vs. Kernel Views

### Memory Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA1000</td>
<td>0</td>
</tr>
<tr>
<td>0xA2000</td>
<td>0</td>
</tr>
<tr>
<td>0xA3000</td>
<td>0</td>
</tr>
<tr>
<td>0xA4000</td>
<td>1</td>
</tr>
<tr>
<td>0xA5000</td>
<td>1</td>
</tr>
<tr>
<td>0xA6000</td>
<td>1</td>
</tr>
<tr>
<td>0xA7000</td>
<td>0</td>
</tr>
<tr>
<td>0xA8000</td>
<td>0</td>
</tr>
<tr>
<td>0xA9000</td>
<td>0</td>
</tr>
<tr>
<td>0xAA000</td>
<td>0</td>
</tr>
</tbody>
</table>

### Diagram

- **Disk**: Data storage
- **Process Memory Segment**:
  - 0x1000
  - 0x2000
  - 0x3000
  - 0xA60B5
## Virtual Memory: User vs. Kernel Views

<table>
<thead>
<tr>
<th>Address</th>
<th>Permissions</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA1000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xA2000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xA3000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xA4000</td>
<td>1X</td>
<td></td>
</tr>
<tr>
<td>0xA5000</td>
<td>1W</td>
<td></td>
</tr>
<tr>
<td>0xA6000</td>
<td>1W</td>
<td></td>
</tr>
<tr>
<td>0xA7000</td>
<td>0W</td>
<td></td>
</tr>
<tr>
<td>0xA8000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xA9000</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0xAA000</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- Disk
- Memory space from 0xA60B5 to 0x1000
- Permissions: R (read), W (write), X (execute)

- 0xA60B5: Read and write permissions
- 0x2000: Read and execute permissions
- 0x3000: Read permission
Virtual Memory: User vs. Kernel Views

write to read-only page ⇒ \textit{segmentation fault}
Virtual Memory: User vs. Kernel Views

jump to non-executable page ⇒ segmentation fault
Virtual Memory: User vs. Kernel Views

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA1000</td>
<td>0</td>
</tr>
<tr>
<td>0xA2000</td>
<td>0</td>
</tr>
<tr>
<td>0xA3000</td>
<td>0</td>
</tr>
<tr>
<td>0xA4000</td>
<td>1 X</td>
</tr>
<tr>
<td>0xA5000</td>
<td>1 W</td>
</tr>
<tr>
<td>0xA6000</td>
<td>1 W</td>
</tr>
<tr>
<td>0xA7000</td>
<td>0 W</td>
</tr>
<tr>
<td>0xA8000</td>
<td>0</td>
</tr>
<tr>
<td>0xA9000</td>
<td>0</td>
</tr>
<tr>
<td>0xAA000</td>
<td>0</td>
</tr>
</tbody>
</table>

Disk
#include "csapp.h"

int main() {
    int x = 8;

    *(int *)&x = 5;
    printf("ok\n");

    *(int *)main = 5;
    printf("not ok\n");

    return 0;
}
Trying to Write to Code Pages

```c
#include "csapp.h"

int main() {
    int x = 8;
    *(int *)&x = 5;
    printf("ok\n");

    *(int *)main = 5;
    printf("not ok\n");
    return 0;
}
```

Fails, because page for `main` is not writable
#include "csapp.h"

int main() {
    /* 0xC3 is the RET instruction */
    char *s1 = "\xC3"
    char *s2 = malloc(1);
    char s3[] = { 0xC3 }

    printf("Trying %p\n", s1);
    ((void (*)())s1)();
    printf("probably ok\n");

    printf("Trying %p\n", s2);
    s2[0] = 0xC3;
    ((void (*)())s2)();
    printf("probably not ok\n");

    printf("Trying %p\n", s3);
    ((void (*)())s3)();
    printf("probably not ok\n");

    return 0;
}
#include "csapp.h"

int main() {
    /* 0xC3 is the RET instruction */
    char *s1 = "\xC3";
    char *s2 = malloc(1);
    char s3[] = { 0xC3 };

    printf("Trying %p\n", s1);
    ((void (*)())s1)();
    printf("probably ok\n");

    printf("Trying %p\n", s2);
    s2[0] = 0xC3;
    ((void (*)())s2)();
    printf("probably not ok\n");

    printf("Trying %p\n", s3);
    ((void (*)())s3)();
    printf("probably not ok\n");

    return 0;
}

Static data tends to be with executable code pages

Other memory is not executable by default
Syscall to Change the Page Table

```c
#include <sys/mman.h>

void *mmap(void *addr, size_t length,
            int prot, int flags,
            int fd, off_t offset);

int munmap(void *addr, size_t length);
```

**mmap** changes the page table:
- **addr** — address to map or NULL for kernel choice
- **length** — bytes to map, rounded up to page size
- **prot** — bitwise PROT_{READ,WRITE,EXEC}
- **flags** — MAP_{PRIVATE,SHARED}, maybe MAP_ANON
- **fd** — file to map into memory if not MAP_ANON
- **offset** — offset into file
Syscall to Change the Page Table

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);
```

Read a file into memory (on demand):

```
fd = open(argv[1], O_RDONLY);
...
p = mmap(NULL, len,
         PROT_READ, MAP_PRIVATE,
         fd, 0);
```
#include <sys/mman.h>

void *mmap(void *addr, size_t length,
           int prot, int flags,
           int fd, off_t offset);

int munmap(void *addr, size_t length);

Allocate a fresh page of memory:

    p = mmap(NULL, getpagesize(),
             PROT_READ | PROT_WRITE,
             MAP_PRIVATE + MAP_ANON,
             -1, 0);

**MAP_ANON with -1 means “not from a file”**
#include "csapp.h"

int main() {
    char *s;
    size_t sz = 1<<14;

    s = Mmap(0, sz,
              PROT_READ | PROT_WRITE | PROT_EXEC,
              MAP_PRIVATE | MAP_ANON,
              -1, 0);

    printf("Trying %p\n", s);
    s[0] = 0xC3;
    ((void (*)())s)();
    printf("ok\n");

    return 0;
}
Changing Page Protection

```c
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

The `mprotect` function changes the protection of previously `mmap`ped pages.
#include "csapp.h"

int main() {
    char *s;
    size_t sz = 1<<14;

    s = Mmap(0, sz,
              PROT_READ | PROT_WRITE,
              MAP_PRIVATE | MAP_ANON,
              -1, 0);
    s[0] = 0xC3;

    Mprotect(s, sz, PROT_READ | PROT_EXEC);

    ((void (*)(()))s)();
    printf("ok\n");

    s[0] = 0x0;
    printf("not ok\n");

    return 0;
}
## Segmentation Fault

Any of these trigger an exception:
- Read of unmapped page
- Write to read-only page
- Jump to non-executable page

Kernel handles the exception by sending a `SIGSEGV` signal

default handler prints “Segmentation Fault” and exits
Handling SIGSEGV

#include "csapp.h"

static char *s;
static size_t sz = 1<<14;

static void recover(int sig) {
    sio_puts("ouch...
    ");
    Mprotect(s, sz, PROT_READ | PROT_WRITE);
}

int main() {
    s = Mmap(0, sz,
              PROT_READ | PROT_EXEC,
              MAP_PRIVATE | MAP_ANON,
              -1, 0);

    Signal(SIGSEGV, recover);

    s[0] = 0x0;
    printf("ok after all\n");

    return 0;
}
Sharing Position-Independent Code

```
0xA1000 0
0xA2000 0
0xA3000 0
0xA4000 0
0xA5000 1
0xA4000 1
0xA7000 0
0xA8000 0
0xA9000 0
0xAA000 0

0x70000 0
0x71000 0
0x72000 0
0x73000 0
0x74000 1
0x73000 1
0x76000 0
0x77000 0
0x78000 0
0x79000 0
```

Disk

image.so

0x1000
0x2000
0x3000
Sharing Position-Independent Code

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA1000</td>
<td>0</td>
</tr>
<tr>
<td>0xA2000</td>
<td>0</td>
</tr>
<tr>
<td>0xA3000</td>
<td>0</td>
</tr>
<tr>
<td>0xA4000</td>
<td>0</td>
</tr>
<tr>
<td>0xA5000</td>
<td>1</td>
</tr>
<tr>
<td>0xA6000</td>
<td>1</td>
</tr>
<tr>
<td>0xA7000</td>
<td>0</td>
</tr>
<tr>
<td>0xA8000</td>
<td>0</td>
</tr>
<tr>
<td>0xA9000</td>
<td>0</td>
</tr>
<tr>
<td>0xAA000</td>
<td>0</td>
</tr>
</tbody>
</table>

0x70000  0
0x71000  0
0x72000  0
0x73000  0
0x74000  1
0x75000  1
0x76000  0
0x77000  0
0x78000  0
0x79000  0
```

Disk

```
0x1000
image.so
  .text
  .plt

0x2000
process 1
  .got
  .got.plt

0x3000
process 2
  .got
  .got.plt
```
Sharing Position-Independent Code

process 1

.process 1

.got

.got.plt

process 2

.process 2

.got

.got.plt

Disk
Virtual Memory and `fork`

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA1000</td>
<td>0</td>
</tr>
<tr>
<td>0xA2000</td>
<td>0</td>
</tr>
<tr>
<td>0xA3000</td>
<td>0</td>
</tr>
<tr>
<td>0xA4000</td>
<td>1</td>
</tr>
<tr>
<td>0xA5000</td>
<td>1</td>
</tr>
<tr>
<td>0xA6000</td>
<td>1</td>
</tr>
<tr>
<td>0xA7000</td>
<td>0</td>
</tr>
<tr>
<td>0xA8000</td>
<td>0</td>
</tr>
<tr>
<td>0xA9000</td>
<td>0</td>
</tr>
<tr>
<td>0xAA000</td>
<td>0</td>
</tr>
</tbody>
</table>

The diagram illustrates the virtual memory space with different permissions for different addresses.
## Virtual Memory and `fork`

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xA1000</td>
<td>0</td>
</tr>
<tr>
<td>0xA2000</td>
<td>0</td>
</tr>
<tr>
<td>0xA3000</td>
<td>0</td>
</tr>
<tr>
<td>0xA4000</td>
<td>1X</td>
</tr>
<tr>
<td>0xA5000</td>
<td>W</td>
</tr>
<tr>
<td>0xA6000</td>
<td>W</td>
</tr>
<tr>
<td>0xA7000</td>
<td>0</td>
</tr>
<tr>
<td>0xA8000</td>
<td>0</td>
</tr>
<tr>
<td>0xA9000</td>
<td>0</td>
</tr>
<tr>
<td>0xAA000</td>
<td>0</td>
</tr>
</tbody>
</table>

Diagram illustrating the memory layout and how `fork` operation affects the memory spaces.
Virtual Memory and `fork`

```c
char a[] = {2, 4, 6, 8};
at 0xA5002
```

![Diagram of virtual memory and fork](chart.png)
Virtual Memory and fork

\[ \text{char } a[] = \{2, 4, 6, 8\}; \quad \text{at } 0xA5002 \]

\[ a[2] = 7 \]
Virtual Memory and fork

```
char a[] = {2, 4, 6, 8};
```

```
a[2] = 7
```
Virtual Memory and \texttt{fork}

\begin{verbatim}
char a[] = \{2, 4, 6, 8\}; at 0xA5002
\end{verbatim}

\begin{verbatim}
a[2] = 7
\end{verbatim}
Virtual Memory and fork

```c
char a[] = {2, 4, 6, 8};
```

```
at 0xA5002
```

```
a[2] = 7
```

```
0xA1000 0
0xA2000 0
0xA3000 0
0xA4000 1 WC
0xA5000 1 WC
0xA6000 1 WC
0xA7000 0
0xA8000 0
0xA9000 0
0xAA000 0
```

```
0xA1000 0
0xA2000 0
0xA3000 0
0xA4000 1 WC
0xA5000 1 WC
0xA6000 1 WC
0xA7000 0
0xA8000 0
0xA9000 0
0xAA000 0
```

```
0x1000 2 4 6 8
```

```
0x2000
```

```
0x3000
```

```
0x4000 2 4 7 8
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```

```
... ...
```
Sharing Pages between Processes

```c
#include "csapp.h"

int main() {
    char *s;
    size_t sz = 1<<14;

    s = Mmap(0, sz,
              PROT_READ | PROT_WRITE | PROT_EXEC,
              MAP_SHARED | MAP_ANON,
              -1, 0);
    s[0] = 1;

    if (Fork() == 0)
        s[0] = 2;
    else
        Wait(NULL);

    printf("%d at %p\n", s[0], s);

    return 0;
}
```
#include "csapp.h"

int main() {
    char *s;
    size_t sz = 1<<14;

    s = Mmap(0, sz,
             PROT_READ | PROT_WRITE | PROT_EXEC,
             MAP_SHARED | MAP_ANON,
             -1, 0);
    s[0] = 1;

    if (Fork() == 0)
        s[0] = 2;
    else
        Wait(NULL);

    printf("%d at %p\n", s[0], s);

    return 0;
}

Using **MAP_SHARED** effectively disables the copy-on-write flag that’s otherwise set by **fork**