The Address-Space Concept

- Protect processes from one another
- Protect the OS from user processes
- Provide efficient management of available storage
Memory Fence

User Area

OS
Base and Bounds Registers

![Diagram showing base and bounds registers]

- Base register
- Bounds
Swapping

User Area

OS
Overlays

Overlay

Resident
Virtual Memory

Process 1

Process 2

Process 3

Memory

Disk
Memory Maps

Virtual Memory

Disk

Memory Map (page table)

Real Memory

pages

Virtual Memory

Disk
Quiz 1

How many $2^{12}$-byte pages fit in a 32-bit address space?

a) a bit over a 1000
b) a bit over a million
c) a bit over a billion
d) none of the above
VM is Your Friend ...

• Not everything has to be in memory at once
  – pages brought in (and pushed out) when needed
  – unallocated parts of the address space consume no memory
    » e.g., hole between stack and dynamic areas

• What’s mine is not yours (and vice versa)
  – address spaces are disjoint

• Sharing is ok though ...
  – address spaces don’t have to be disjoint
    » a single page frame may be mapped into multiple processes

• I don’t trust you (or me)
  – access to individual pages can be restricted
    » read, write, execute, or any combination
Page-Table Size

- Consider a full $2^{32}$-byte address space
  - assume 4096-byte ($2^{12}$-byte) pages
  - 4 bytes per page-table entry
  - the page table would consist of $2^{32}/2^{12} (= 2^{20})$ entries
  - its size would be $2^{22}$ bytes (or 4 megabytes)
    » at $100$/gigabyte
    • around $0.40$

- For a $2^{64}$-byte address space
  - assume 4096-byte ($2^{12}$-byte) pages
  - 8 bytes per page-table entry
  - the page table would consist of $2^{64}/2^{12} (= 2^{52})$ entries
  - its size would be $2^{55}$ bytes (or 32 petabytes)
    » at $1$/gigabyte
    • over $33$ million
IA32 Paging

10 bits

Page directory table

10 bits

Page table

12 bits

Page

CR3
Quiz 2

Can a page start at a virtual address that’s not divisible by the page size?

a) yes
b) no
Linux Intel IA32 VM Layout

Page directory table

kernel

user

0

3GB

4GB
x86-64 Virtual Address Format 1

63  47  38  29  20  11  0

unused

Page map table

Page directory
pointer table

Page directory

Page table

4KB page
x86-64 Virtual Address Format 2

- Page map table
- Page directory pointer table
- Page directory table
- 2MB page
Why Multiple Page Sizes?

- **External fragmentation**
  - for region composed of 4KB pages, average external fragmentation is 2KB
  - for region composed of 1GB pages, average external fragmentation is 512MB

- **Page-table overhead**
  - larger page sizes have fewer page tables
    » less overhead in representing mappings
Address Space

- **OS kernel**: 0xffffffffffffffff
- **Illegal**: 0xffffffff80000000000000
  - 2^{47} bytes
- **User**: 0x00007fffffffffff
  - 2^{47} bytes
- **Illegal**: 0x0000800000000000
  - 2^{64} – 2^{48} bytes
Performance

• Page table resides in real memory (DRAM)
• A 32-bit virtual-to-real translation requires two accesses to page tables, plus the access to the ultimate real address
  – three real accesses for each virtual access
  – 3X slowdown!
• A 64-bit virtual-to-real translation requires four accesses to page tables, plus the access to the ultimate real address
  – 5X slowdown!
Translation Lookaside Buffers

<table>
<thead>
<tr>
<th>Tag</th>
<th>Key</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

- **Tag**: Indicators of specific data entries.
- **Key**: Used for quick access.
- **Offset**: Additional memory location.
Quiz 3

Recall that there is a 5x slowdown on memory references via virtual memory on the x86-64. If all references are translated via the TLB, the slowdown will be

a) 1x
b) 2x
c) 3x
d) 4x
OS Role in Virtual Memory

• **Memory is like a cache**
  - quick access if what’s wanted is mapped via page table
  - slow if not — OS assistance required

• **OS**
  - make sure what’s needed is mapped in
  - make sure what’s no longer needed is not mapped in
Mechanism

- Program references memory
  - if reference is mapped, access is quick
    » even quicker if translation in TLB and referent in on-chip cache
  - if not, page-translation fault occurs and OS is invoked
    » determines desired page
    » maps it in, if legal reference
Issues

• Fetch policy
  – when are items put in the cache?

• Placement policy
  – where do they go in the cache?

• Replacement policy
  – what’s removed to make room?
Hardware Caches

• Fetch policy
  – when are items put in the cache?
    » when they’re referenced
    » prefetch might be possible (e.g., for sequential access)

• Placement policy
  – where do they go in the cache?
    » usually determined by cache architecture
    » if there’s a choice, it’s typically a random choice

• Replacement policy
  – what’s removed to make room?
    » usually determined by cache architecture
    » if there’s a choice, it’s typically a random choice
Software Caches

• Fetch policy
  – when are items put in the cache?
    » when they’re referenced
    » prefetch might be easier than for hardware caches

• Placement policy
  – where do they go in the cache?
    » usually doesn’t matter (no memory is more equal than others)

• Replacement policy
  – what’s removed to make room?
    » would like to remove that whose next use is farthest in future
    » instead, remove that whose last reference was farthest in the past
The “Pageout Daemon”

- In-Use Page Frames
- Pageout Daemon
- Disk
- Free Page Frames
Managing Page Frames
Clock Algorithm

Back hand:
if (reference bit == 0)
remove page

Front hand:
reference bit = 0
Why is virtual memory used?
More VM than RM
Isolation

Virtual Memory

Process 1

0
1
2
3
4
5

Memory Maps (page tables)

Process 2

0
1
2
3
4
5

Real Memory

page frames

0
1
2
3
4
5
6
7

0
1
2
3
4
5
6
7
Virtual Memory

Sharing

Process 1

Memory Maps (page tables)

Real Memory

page frames

Virtual Memory
File I/O

Buffer

User Process

Buffer Cache
Multi-Buffered I/O

Process

\texttt{read( ... )}

\texttt{i-1}  previous block

\texttt{i}  current block

\texttt{i+1}  probable next block
Traditional I/O

User Process 1:
1: read f1, p0
3: read f1, p1
5: read f3, p0

User Process 2:
2: read f2, p0
4: read f2, p1
5: read f3, p0

Disk

File 1
- page 0
- page 1
- page 0
Buffer Cache
- page 0
- page 1
- page 0
Kernel Memory
- page 0
- page 1
- page 0
- page 0

File 2
- page 0
- page 1
- page 2
- page 3
- page 4
- page 5
- page 6
- page 7

File 3
- page 0
- page 1
- page 0
- page 0
- page 0
- page 0
- page 0
- page 0
Mapped File I/O

Process 1
Virtual Memory

Real Memory

Disk

File 1

page 0

page 1

page 2

page 3

page 4

page 5

page 6

page 7
Multi-Process Mapped File I/O

Process 2
Virtual Memory
CS33 Intro to Computer Systems

Real Memory

Disk

File 1

page 0
page 1
page 2
page 3
page 4
page 5
page 6
page 7
Mapped Files

- **Traditional File I/O**
  ```
cchar buf[BigEnough];
fd = open(file, O_RDWR);
for (i=0; i<n_recs; i++) {
    read(fd, buf, sizeof(buf));
    use(buf);
}
```

- **Mapped File I/O**
  ```
record_t *MappedFile;
fd = open(file, O_RDWR);
MappedFile = mmap(..., fd, ...);
for (i=0; i<n_recs; i++)
    use(MappedFile[i]);
```
Mmap System Call

```c
void *mmap(
    void *addr,
    // where to map file (0 if don’t care)
    size_t len,
    // how much to map
    int prot,
    // memory protection (read, write, exec.)
    int flags,
    // shared vs. private, plus more
    int fd,
    // which file
    off_t off
    // starting from where
);
```
The *mmap* System Call

![Diagram of mmap system call](image)
Share-Mapped Files

Data = 17;
Private-Mapped Files

L1 Page Table

L2 Page Tables

File Pages

L2 Page Tables

L1 Page Table

Data = 17;
Example

```c
int main( ) {
    int fd;
dataObject_t *dataObjectp;

    fd = open("file", O_RDWR);
    if ((int)(dataObjectp = (dataObject_t *)mmap(0,
        sizeof(dataObject_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == -1) {
        perror("mmap");
        exit(1);
    }

    // dataObjectp points to region of (virtual) memory
    // containing the contents of the file

    ...
}
```
fork and mmap

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

    if (fork() == 0) {
        // in child
        x = 2;
        exit(0);
    }
    // in parent
    while (x==1) {
        // will loop forever
    }
    return 0;
}
```

```c
int main() {
    int fd = open( ... );
    int *xp = (int *)mmap(...,
                        MAP_SHARED, fd, ...);
    xp[0] = 1;
    if (fork() == 0) {
        // in child
        xp[0] = 2;
        exit(0);
    }
    // in parent
    while (xp[0]==1) {
        // will terminate
    }
    return 0;
}
```