Virtualization and the UNIX API

CS 450: Operating Systems
Sean Wallace <swallac6@iit.edu>
Agenda

- The Process
  - UNIX process management API
- Virtual Memory
- Dynamic memory allocation & related APIs
The Process
Definition & OS Responsibilities
Process = a program in execution
{ code (program),
  global data,
  local data (stack),
  dynamic data (heap),
  PC & other registers }
Programs *describe* what we want done,
Processes *realize* what we want done
...and the operating system *runs* processes
Execution = running a process

Scheduling = running *many* processes

Peripherals = I/O devices
To do this, the OS is constantly running “in the background”, keeping track of a large amount of process/system metadata.
{ code (program),
  global data,
  local data (stack),
  dynamic data (heap),
  PC & other registers,
  + OS-level metadata }
\{ \text{code (program),} \\
\text{global data,} \\
\text{local data (stack),} \\
\text{dynamic data (heap),} \\
\text{PC \& other registers,} \\
\text{+ (e.g., pid, owner, memory/CPU usage)} \}
Logical Control Flow

Process A  Process B  Process C

Time
Physical Flow (1 CPU)
Context Switches

Time

Process A

<table>
<thead>
<tr>
<th>Disk interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return from read</td>
</tr>
<tr>
<td>read</td>
</tr>
</tbody>
</table>

Process B

| User Code |
| Kernel Code |
| User Code |
| Kernel Code |
| User Code |

Context switch
Context switches are external to a process’s logical control flow (dictated by user program)

—part of key OS virtualization:

\textit{exceptional} control flow
Exceptional Control Flow
int main() {
    while (1) {
        printf("Hello World!\n");
    }
    return 0;
}
Two classes of exceptions:

I. Synchronous

II. Asynchronous
1. **Synchronous exceptions** are caused by the *currently executing* instruction
3 subclasses of synchronous exceptions:

1. Traps
2. Faults
3. Aborts
1. Traps

Traps are intentionally triggered by a process

E.g., to invoke a system call
char *str = "Hello World"
int len = strlen(str)
write(1, str, len);

mov edx, len
mov ecx, str
mov ebx, 1
mov eax, 4    ; syscall #4
int 0x80      ; trap to OS
Return from trap (if it happens) resumes execution at the next logical instruction
1. Faults

Faults are usually *unintentional*, and may be recoverable or irrecoverable.

E.g., segmentation fault, protection fault, page fault, divide-by-zero.
Often, return from fault will result in *retrying* the faulting instruction

— Esp. if the handler “fixes” the problem
1. Aborts

Aborts are *unintentional* and *irrecoverable*

I.e., abort = program/OS termination

E.g., memory ECC error
I. **Asynchronous exceptions** are caused by events *external to* the current instruction
int main() {
    while (1) {
        printf("Hello World!\n");
    }
    return 0;
}
Hardware initiated asynchronous exceptions are known as *interrupts*
E.g., ctrl-C, ctrl-alt-del, power switch
Interrupts are associated with specific processor (hardware) pins

- Checked after every CPU cycle
- Associated with interrupt handlers
(system) memory

interrupt vector

int. handler 0 code

int #
• (Typical) interrupt procedure
  • Save context (e.g., user process)
  • Load OS context
  • Execute handler
  • Load context (for…?)
  • Return
Important: after switching context to the OS (for exception handling), there is no guarantee if/when a process will be switched back in!
Switching context to the kernel is potentially very expensive

— But…it’s the only way to the OS API!
UNIX API
Process Management
Creating Processes
#include <unistd.h>

pid_t fork();
fork traps to OS to create a new process

... which is (mostly) a duplicate of the calling process!
• E.g., the new (child) process runs the same program as the creating (parent) process

• And starts with the same PC,

• The same SP, FP, regs,

• The same open files, etc., etc.
Parent

```c
int main() {
    fork();
    foo();
}
```
44

OS

creates

\[ P_{parent} \]
\[
\text{int main() \{} \\
\text{fork();} \\
\text{foo();} \\
\text{\}} \\
\]

\[ P_{child} \]
\[
\text{int main() \{} \\
\text{fork();} \\
\text{foo();} \\
\text{\}} \\
\]
int main() {
    fork();
    foo();
}

int main() {
    fork();
    foo();
}
fork, when called, returns twice

(to each process @ the next instruction)
System-wide unique process identifier

Child’s pid (> 0) is returned in the parent

Sentinel value 0 is returned in the child

```c
typedef int pid_t;

pid_t fork();
```
```c
void fork0() {
    int pid = fork();
    if (pid == 0) {
        printf("Hello from Child!\n");
    }
    else {
        printf("Hello from Parent!\n");
    }
}

main() { fork0(); }
```

Hello from Child!
Hello from Parent!

(or)

Hello from Parent!
Hello from Child!
• That is, order of execution is *nondeterministic*

• Parent & child run *concurrently!*
void fork1() {
    int x = 1;

    if (fork() == 0) {
        printf("Child has x = %d\n", ++x);
    }
    else {
        printf("Parent has x = %d\n", --x);
    }
}
• Important: post-fork, parent & child are identical, but separate!

• OS allocates and maintains separate data/state

• Control flow can diverge
All terminating processes turn into zombies
• “Dead” but still tracked by OS
• PID remains in use
• Exit status can be queried
All processes are responsible for *reaping* their own (immediate) children
pid_t wait(int *stat_loc);
pid_t wait(int *stat_loc);

• When called by a process with ≥ 1 children:
  
  • *Waits* (if needed) for a child to terminate
  
  • *Reaps* a zombie child (if ≥ 1 zombified children, arbitrarily pick one)
  
  • *Returns* reaped child’s PID and exit status info via pointer (if non-NULL)
wait allows us to synchronize one process with events (e.g., termination) in another
void fork10() {
    int i, stat;
    pid_t pid[5];
    for (i = 0; i < 5; i++) {
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            exit(100 + i);
        }
    }
    for (i = 0; i < 5; i++) {
        pid_t cpid = wait(&stat);
        if (WIFEXITED(stat)) {
            printf("Child %d terminated with status %d\n", cpid, WEXITSTATUS(stat));
        }
    }
}
/* explicit waiting -- i.e., for a specific child */
pid_t waitpid(pid_t pid, int *stat_loc, int options);

/** Wait options **/

/* return 0 immediately if no terminated children */
#define WNOHANG 0x00000001

/* also report info about stopped children (and others) */
#define WUNTRACED 0x00000002
```c
void fork11() {
    int i, stat;
    pid_t pid[5];
    for (i = 0; i < 5; i++) {
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            exit(100 + i);
        }
    }
    for (i = 0; i < 5; i++) {
        pid_t cpid = waitpid(pid[i], &stat, 0);
        if (WIFEXITED(stat)) {
            printf("Child %d terminated with status %d\n", cpid, WEXITSTATUS(stat));
        }
    }
}
```

Child 9085 terminated with status 100
Child 9086 terminated with status 101
Child 9087 terminated with status 102
Child 9088 terminated with status 103
Child 9089 terminated with status 104
```c
int main(int argc, const char * argv[]) {
    int stat;
    pid_t cpid;

    if (fork() == 0) {
        printf("Child pid = %d\n", getpid());
        sleep(3);
        exit(1);
    }
    else {
        /* use with -1 to wait on any child (with options) */
        while ((cpid = waitpid(-1, &stat, WNOHANG)) == 0) {
            sleep(1);
            printf("No terminated children!\n");
        }
        printf("Reaped %d with exit status %d\n", cpid, WEXITSTATUS(stat));
    }
}
```

Child pid = 9108
No terminated children!
No terminated children!
No terminated children!
Reaped 9108 with exit status 1
• Recap:
  • fork: create new (duplicate) process
  • wait: reap terminated (zombie) process
Running *new programs* (within processes)
/* the "exec family" of syscalls */

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
Execute a *new program* within the *current process context*
Complements \texttt{fork} (1 call $\rightarrow$ 2 returns):

- When called, \texttt{exec} (if successful) never returns!
- Starts execution of new program
int main(int argc, const char * argv[]) {
    execl("/bin/echo", "bin/echo", "hello", "world", (void *)0);
    printf("Done exec-ing...\n");
    return 0;
}

$ ./a.out
hello world
```c
int main(int argc, const char * argv[]) {
    printf("About to exec!\n");
sleep(1);
    execl("./execer", ".\execer", (void *)0);
    printf("Done exec-ing...\n");
    return 0;
}
```

```bash
$ gcc execer.c -o execer
$ ./execer
About to exec!
About to exec!
About to exec!
About to exec!
About to exec!
...
```
```c
int main(int argc, const char * argv[]) {
    if (fork() == 0) {
        execl("/bin/ls", "/bin/ls", "-l", (void *)0);
        exit(0); /* in case exec fails */
    }
    wait(NULL);
    printf("Command completed\n");
    return 0;
}
```

```
$ ./a.out
-rwxr-xr-x  1 sean  staff  9500 Oct 7 00:15 a.out
Command completed
```
Interesting question:

Why are fork & exec separate syscalls?

/* i.e., why not: */
fork_and_exec("/bin/ls", ...)


A1: we might really want to just create duplicates of the current process (e.g.?)
A2: we might want to replace the current program without creating a new process
A3 (more subtle): we might want to “tweak” a process *before* running a program in it
The Unix Family Tree
BIOS

bootloader

kernel

“handcrafted” process
fork & exec

kernel

init

fork & exec

“Daemons”
e.g., sshd, http

getty
kernel \rightarrow \text{init} \rightarrow \text{shell (e.g., sh)} \rightarrow \text{exec}
kernel → init → shell (e.g., sh) → user process → user process → user process → user process

(a fork-ing party!)
(or, for the GUI-inclined)

- **kernel**
- **init**
- **display manager (e.g., xdm)**
- **X Server (e.g., XFree86)**
- **window manager (e.g., twm)**
window manager (e.g., twm)

terminal emulator (e.g., xterm)

shell (e.g., sh)

user process

user process

user process

user process
The Shell (aka the CLI)
The original operating system user interface
Essential function: let the user issue requests to the operating system

e.g., fork/exec a program,

manage processes (list/stop/term),
browse/manipulate the file system
(a read-eval-print-loop (REPL) for the OS)
pid_t pid;
char buf[80], *argv[10];

while (1) {
    /* print prompt */
    printf("$ ");

    /* read command and build argv */
fgets(buf, 80, stdin);
    for (i = 0, argv[0] = strtok(buf, " \n");
        argv[i];
        argv[++i] = strtok(NULL, " \n");

    /* fork and run command in child */
    if ((pid = fork()) == 0) {
        if (execvp(argv[0], argv) < 0) {
            printf("Command not found\n");
            exit(0);
        }
    }
}

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
The process management API (fork, wait, exec, etc.) let us tap into CPU *virtualization*
i.e., *create, manipulate, clean up* concurrently running programs
Next key component in the Von Neumann machine: *memory*

To be accessed by all of our processes…

- Simultaneously

- With a simple, consistent API

- Withint trampling on each other!
Virtual Memory
Motivation
Again, recall the Von Neumann architecture—a stored-program computer with programs and data stored in the same memory.
“Memory” is an *idealized* storage device that hold our programs (instructions) and data (operands)
Colloquially: “RAM”, *random access memory*

~ big array of byte-accessible data
execlp("/bin/echo", "/bin/echo", "hello", NULL);

but our code & data clearly reside on the hard drive (to start)!
in reality, “memory” is a combination of storage systems with very different access characteristics
common types of “memory”:

SRAM, DRAM, NVRAM, HDD
# Relative Speeds

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Access latency</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>8 - 32 words</td>
<td>0 - 1 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>On-board SRAM</td>
<td>32 - 256 KB</td>
<td>1 - 3 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>Off-board SRAM</td>
<td>256 KB - 16 MB</td>
<td>~10 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>DRAM</td>
<td>128 MB - 64 GB</td>
<td>~100 cycles</td>
<td>(ns)</td>
</tr>
<tr>
<td>SSD</td>
<td>≤ 1 TB</td>
<td>~10,000 cycles</td>
<td>(µs)</td>
</tr>
<tr>
<td>HDD</td>
<td>≤ 4 TB</td>
<td>~10,000,000 cycles</td>
<td>(ms)</td>
</tr>
</tbody>
</table>
“Numbers Every Programmer Should Know”
http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html
would like:

1. a lot of memory
2. fast access to memory
3. to not spend $$$ on memory
an exercise in compromise: *the memory hierarchy*
idea: use the fast but scarce kind as much as possible; fall back on the slow but plentiful kind when necessary
focus on DRAM ⇔ HDD, SSD, etc.
i.e., memory as a “cache” for disk
main goals:

1. maximize memory throughput
2. maximize memory utilization
3. provide address space consistency & memory protection to processes
throughput = # bytes per second

- depends on access latencies (DRAM, HDD) and “hit rate”

- improve by minimizing disk accesses
utilization = fraction of allocated memory that contains “user” data (aka payload)

- reduced by storing metadata in memory
- affected by alignment, block size, etc.
address space consistency → provide a uniform “view” of memory to each process
address space consistency → provide a uniform “view” of memory to each process
memory protection $\rightarrow$ prevent processes from directly accessing each other’s address space
memory protection → prevent processes from directly accessing each other’s address space
i.e., every process should be provided with a managed, *virtualized* address space
“memory addresses”: what are they, really?
“physical” address: (byte) index into DRAM
int glob = 0xDEADBEEE;

main() {
    fork();
    glob += 1;
}

(gdb) set detach-on-fork off
(gdb) break main
Breakpoint 1 at 0x400508: file memtest.c, line 7.
(gdb) run
Breakpoint 1, main () at memtest.c:7
7    fork();
(gdb) next
[New process 7450]
8    glob += 1;
(gdb) print &glob
$1 = (int *) 0x6008d4
(gdb) next
9   }
(gdb) print /x glob
$2 = 0xdeadbeef
(gdb) inferior 2
[Switching to inferior 2 [process 7450]
#0 0x0000000310acac49d in __libc_fork ()
131    pid = ARCH_FORK ();
(gdb) finish
Run till exit from #0 in __libc_fork ()
8    glob += 1;
(gdb) print /x glob
$4 = 0xdeadbeee
(gdb) print &glob
$5 = (int *) 0x6008d4
instructions executed by the CPU do not refer directly to physical addresses!
processes reference *virtual* addresses, the CPU relays virtual address requests to the *memory management unit* (MMU), which are *translated* to physical addresses.
The diagram illustrates the memory management unit (MMU) and the relationship between virtual and physical addresses.

- The CPU generates a virtual address.
- The MMU translates the virtual address to a physical address.
- The physical address is sent to Main Memory.
- If Main Memory does not contain the requested data, it is retrieved from "swap" space on disk.

(Note: cache not shown)
the size of virtual memory is determined by the virtual address width

the physical address width is determined by the amount of installed physical memory
e.g., given 48-bit virtual address, 8GB installed DRAM
- 48-bits $\rightarrow$ 256TB virtual space
- 8GB $\rightarrow$ 33-bit physical address
$P_0$

virtual address space

virtual address

$P_1$

virtual address space

virtual address

$P_2$

virtual address space

virtual address

(physical) address space

physical address

$2^{n-1}$

$2^{m-1}$
essential problem: map request for a virtual address $\rightarrow$ physical address

… and this must be **FAST!** (happens on every memory access)
both hardware/software are involved:

- MMU (hw) handles simple and fast operations (e.g., table lookups)

- Kernel (sw) handles complex tasks (e.g., eviction policy)
Virtual Memory Implementation
keep in mind goals:

1. maximize memory throughput
2. maximize memory utilization
3. provide address space consistency & memory protection to processes
1. simple relocation

- per-process relocation address is loaded by kernel on every context switch
1. simple relocation

- incorporate a limit register to provide memory protection
pros:

- simple & fast!
- provides protection
but: available memory for mapping depends on value of base address

i.e., address spaces are not consistent!
also: all of a process *below the address limit* must be loaded in memory

i.e., memory may be *vastly under-utilized*
2. segmentation

- partition virtual address space into *multiple logical segments*

- individually map them onto physical memory with relocation registers
virtual address has form \texttt{seg#::offset}
MMU
Segment Table

<table>
<thead>
<tr>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>$L_0$</td>
</tr>
<tr>
<td>$B_1$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>$B_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$B_3$</td>
<td>$L_3$</td>
</tr>
</tbody>
</table>

CPU ➔ VA: **seg#:offset**

$\text{PA}: \text{offset} + B_2$

assert ($\text{offset} \leq L_2$)
- implemented as MMU registers

- part of kernel-maintained, per-process metadata (aka “process control block”)

- re-populated on each context switch
pros:
- still very fast
  - translation = register access & addition
- memory protection via limits
- segmented addresses improve consistency
what about utilization?
simple relocation:

segmentation:
- variable segment sizes → memory fragmentation
- fragmentation potentially lowers utilization
- can fix through compaction, but expensive!
3. paging

- partition virtual and physical address spaces into *uniformly sized pages*

- only map pages onto physical memory that contain required data
- pages boundaries are *not aligned to segments*!
- instead, aligned to multiples of page size
- minimum mapping granularity = page
- not all of a given segment need be mapped
new mapping problem:

- given virtual address, decompose into 
  \textit{virtual page number} \& \textit{virtual page offset}

- map VPN $\rightarrow$ \textit{physical page number}
Given page size = $2^p$ bytes

**VA:**

<table>
<thead>
<tr>
<th>virtual page number</th>
<th>virtual page offset</th>
</tr>
</thead>
</table>

**PA:**

<table>
<thead>
<tr>
<th>physical page number</th>
<th>physical page offset</th>
</tr>
</thead>
</table>
VA: virtual page number virtual page offset

address translation

PA: physical page number physical page offset
### Translation Structure: Page Table

- **VA:** virtual page number
- **PA:** physical page number
- **valid**
- **PPN**
- **index**

If invalid, page is not mapped.

- **n**
- **2^n entries**
page table entries (PTEs) typically contain additional metadata, e.g.: 

- dirty (modified) bit 
- access bits (shared or kernel-owned pages may be read-only or inaccessible)
e.g., 32-bit virtual address, 4KB \( (2^{12}) \) pages, 4-byte PTEs;
- size of page table?
e.g., 32-bit virtual address,
4KB ($2^{12}$) pages,
4-byte PTEs;

- # pages $= 2^{(32-12)} = 2^{20} = 1M$

- page table size $= 1M \times 4$ bytes $= 4MB$
4MB is much too large to fit in the MMU — insufficient registers and SRAM!

Page table resides in **main memory**
The translation process (aka *page table walk*) is performed by hardware (MMU).

The kernel must initially populate, then continue to manage a process’s page table.

The kernel also populates a *page table base register* on context switches.
translation: *hit*

1. VA: $N$
2. *page table walk*
3. PA: $N'$
4. *data*

CPU → Address Translator (part of MMU) → Page Table → Main Memory

1. VA: $N$
2. *page table walk*
3. PA: $N'$
4. *data*
translation: *miss*

1. VA: $N$
2. page table walk
3. page fault
4. transfer control to kernel
5. data transfer
6. PTE update
7. VA: $N$ (retry)
8. walk
9. PA: $N'$
10. data
kernel decides where to place page, and what to evict (if memory is full)

- e.g., using LRU replacement policy
this system enables on-demand paging
i.e., an active process need only be partly in memory (load rest from swap dynamically)
but if working set (of active processes) exceeds available memory, we may have *swap thrashing*
integration with caches?
Q: Do caches use physical or virtual address for lookups?
Virtual Address Based Cache

Process A

Virtual Address Space

| M | L | 0 | X |

Process B

Virtual Address Space

| N | M | 0 | Z |

“Synonym” problem
Physical Address Based Cache

Process A

Virtual Address Space

Virtual Address Space

Process B

CPU

Cache

Address | Data
--- | ---
S | X
Q | Y
R | Z

Physical Memory

Physical Memory

158

IIT College of Science

ILLINOIS INSTITUTE OF TECHNOLOGY
Q: Do caches use physical or virtual address for lookups?

A: Caches typically use physical addresses
CPU

MMU (address translation unit)

Cache

Main Memory

process page table

page table walk

%*@$&#!

160
Saved by hardware:

The *Translation Lookaside Buffer* (TLB) — a cache used solely for VPN→PPN lookups
page table walk

MMU (address translation unit)

Cache

Main Memory

process page table

CPU

VA

(data)

PA

(hit)

(update)

(data)

(hit)

(update)

(miss)
only if TLB miss!

TLB + Page Table
(exercise for you: revise earlier translation diagrams!)
virtual address

n-1 virtual page number (VPN) p p-1 page offset

valid tag physical page number (PPN)

TLB

valid tag data

Cache

byte offset

Cache Hit

TLB Hit
Not so fast!

TLB mappings are *process specific*—requires flush & reload on context switch

- Some architectures store PID (aka “virtual space” ID) in TLB
• Another problem:
  • TLB caches a few thousand mappings
  • vs. *millions* of virtual pages per process!
We can improve TLB hit rate by reducing the number of pages...

by increasing the size of each page.
• Compute # of pages for 32-bit memory for:
  
  • 1KB, 512KB, 4MB pages
    
    • \(2^{32} \div 2^{10} = 2^{22} = 4\text{M pages}\)
    
    • \(2^{32} \div 2^{19} = 2^{13} = 8\text{K pages}\)
    
    • \(2^{32} \div 2^{22} = 2^{10} = 1\text{K pages} \) (not bad!)
Virtual Memory

Process A

Physical Memory

Lots of wasted space!

Physical Memory

Process B

Virtual Memory
Virtual Memory

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
</table>

Process A

Virtual Memory

<table>
<thead>
<tr>
<th>Physical Memory</th>
</tr>
</thead>
</table>

Process B

Virtual Memory
Increasing page size results in increased internal fragmentation and lower utilization
That is, TLB effectiveness needs to be balanced against memory utilization!
So, what about 64-bit systems?

$2^{64} = 16 \text{ ExbiByte address space}$

$\approx 4 \text{ billion } \times 4 \text{ GB}$
Most modern implementations support a max of $2^{48}$ (256TB) addressable space.
• Page table size?

• # pages $= 2^{48} \div 2^{12} = 2^{36}$

• PTE size $= 8$ bytes (64 bits)

• PT size $= 2^{36} \times 8 = 2^{39}$ bytes
  $= 512$GB
512GB

(just for the virtual memory *mapping*
structure)

(and we need *one per process*)
We’re not going to be able to fit these into memory
• Instead, use *multi-level* page tables:
  
  • Split an address translation into two (or more) separate table lookups
  
  • Unused parts of the table don’t need to be in memory!
“Toy memory system”
- 8 bit addresses
- 32-byte pages

All 8 PTEs must be in memory at all times
“Toy memory system”
- 8 bit addresses
- 32-byte pages

page offset

Page "directory"
all unmapped; don’t need in memory!
“Toy memory system”
- 8 bit addresses
- 32-byte pages
Intel Architecture Memory Management

http://www.intel.com/products/processor/manuals/

(Software Developer’s Manual Volume 3A)
### Segment Descriptors

<table>
<thead>
<tr>
<th>Segment Registers</th>
<th>Segment Descriptors</th>
<th>Linear Address Space (or Physical Memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>Access</td>
<td>Limit</td>
</tr>
<tr>
<td>SS</td>
<td>Access</td>
<td>Limit</td>
</tr>
<tr>
<td>DS</td>
<td>Access</td>
<td>Limit</td>
</tr>
<tr>
<td>ES</td>
<td>Access</td>
<td>Limit</td>
</tr>
<tr>
<td>FS</td>
<td>Access</td>
<td>Limit</td>
</tr>
<tr>
<td>GS</td>
<td>Access</td>
<td>Limit</td>
</tr>
</tbody>
</table>
“Flat” Model
## Paging Modes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>32</td>
<td>32</td>
<td>N/A</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>32-bit</td>
<td>1</td>
<td>0</td>
<td>0^2</td>
<td>32</td>
<td>Up to 40^3</td>
<td>4 KB, 4 MB^4</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PAE</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>Up to 52</td>
<td>4 KB, 2 MB</td>
<td>Yes^5</td>
<td>No</td>
</tr>
<tr>
<td>IA-32e</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>48</td>
<td>Up to 52</td>
<td>4 KB, 2 MB, 1 GB^6</td>
<td>Yes^5</td>
<td>Yes^7</td>
</tr>
</tbody>
</table>
32-Bit Paging (4KB Pages)
32-Bit Paging (4MB Pages)
IA-32e Paging (4KB Pages)
IA-32e Paging (1GB Pages)
Dynamic Memory Allocation
From:

The Memory Hierarchy

- registers
- cache (SRAM)
- main memory (DRAM)
- local hard disk drive (HDD/SSD)
- remote storage (networked drive / cloud)
We now have:

Virtual Memory
Now what?
Static Data

- code, global variables, jump tables, etc.
- Allocated at fork/exec
- Lifetime: permanent
The Stack

- Function activation records
  - Local vars, arguments, return values
  - Lifetime: LIFO

_pages allocated as needed (up to preset stack limit)
Explicitly requested from the kernel

- For **dynamic allocation**
- Lifetime: *arbitrary!*

The *Heap*
- Starts out empty
- \texttt{brk} pointer marks top of the heap

The \textit{Heap}
Heap mgmt syscall:

```c
void *sbrk(int inc); /* resizes heap by inc, returns old brk value */
```

The Heap
void *hp = sbrk(N);

The Heap
Can use `sbrk` to allocate structures:

```c
int **make_jagged_arr(int nrows, const int *dims) {
    int i, j;
    int **jar = sbrk(sizeof(int *) * nrows);

    for (i = 0; i < nrows; i++) {
        jarr[i] = sbrk(sizeof(int) * dims[i]);
    }

    return jarr;
}
```
But, we can’t “free” this memory!

```c
int **make_jagged_arr(int nrows, const int *dims) {
    int i, j;
    int **jar = sbrk(sizeof(int *) * nrows);
    for (i = 0; i < nrows; i++) {
        jar[i] = sbrk(sizeof(int) * dims[i]);
    }
    return jar;
}

void free_jagged_arr(int **jar, int nrows) {
    int i;
    for (i = 0; i < nrows; i++) {
        free(jar[i]);
    }
    free(jar);
}
```
After the kernel allocates heap space for a process, it is *up to the process to manage it!*
“Manage” = tracking memory in use, tracking memory not in use, reusing unused memory
Job of the *dynamic memory allocator*

— Typically included as a user-level library and/or language runtime feature
User Process

- application program
- dynamic memory allocator

malloc

Heap

sbrk

OS Kernel

malloc

Heap

RAM

Disk
User Process

application program

dynamic memory allocator

free(p)

Heap

OS Kernel

RAM

Disk
User Process

application program

dynamic memory allocator

heap space may not be returned to the kernel!

free(p)

Heap

OS Kernel

RAM

Disk
The DMA constructs a *user-level* abstraction (re-usable “blocks” of memory) *on top of a kernel-level* one (virtual memory)
The user-level implementation must make good use of the underlying infrastructure (the memory hierarchy)