Virtualization and the UNIX API

CS 450: Operating Systems
Michael Saelee <lee@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 regs }
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,
  global data,
  local data (stack),
  dynamic data (heap),
  PC & other regs,
  + OS-level metadata }
code,
global data,
local data (stack),
dynamic data (heap),
PC & other regs,
+ (e.g., pid, owner, memory/CPU usage)
Logical control flow
Physical flow (1 CPU)
Context switches
context switches are external to a process’s logical control flow (dictated by user program) — part of key OS virtualization: exceptional control flow
§ Exceptional Control Flow
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
Two classes of exceptions:

I. synchronous

II. asynchronous
I. 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
2. faults

faults are usually *unintentional*, and may be recoverable or irrecoverable

e.g., segmentation fault, protection fault, page fault, div-by-zero
often, return from fault will result in
*retrying* the faulting instruction
— esp. if the handler “fixes” the problem
3. aborts

Abort is unintentional and irrecoverable. i.e., abort = program/OS termination. e.g., memory ECC error.
II. asynchronous exceptions are caused by events external to the current instruction
```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```

```
hello world!
hello world!
hello world!
hello world!
^C
$
```
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 #
  - int. handler 0 code
interrupt procedure (typical)
- 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 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

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

OS`
P_{\text{parent}}\begin{center}
\begin{verbatim}
int main () {
    fork();
    foo();
}
\end{verbatim}
\end{center}
```c
int main () {
    fork();
    foo();
}
```

This diagram illustrates the concept of a program creating a child process, which is a common operation in operating systems (OS). The parent process (`P_{parent}`) calls the `fork()` function to create a child process (`P_{child}`). Both processes contain the same code, as shown in the diagram.
```c
int main () {
    fork();
    foo();
}
```

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

(to each process @ the next instruction)
typedef int pid_t;

pid_t fork();

- system-wide unique process identifier
- child’s pid (> 0) is returned in the parent
- sentinel value 0 is returned in the child
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!
i.e., order of execution is *nondeterministic*

- parent & child run *concurrently*!
```c
void fork1 () {
    int x = 1;

    if (fork() == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
}
```

Parent has x = 0
Child has x = 2
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 \( \geq 1 \) children:

- waits (if needed) for a child to terminate
- reaps a zombie child (if \( \geq 1 \) 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));
    }
}

Child 8590 terminated with status 101
Child 8589 terminated with status 100
Child 8593 terminated with status 104
Child 8592 terminated with status 103
Child 8591 terminated with status 102
/* 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
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));
    }
}
```c
int main() {
    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 = 8885
No terminated children!
No terminated children!
No terminated children!
Reaped 8885 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 `fork` (1 call → 2 returns):

- when called, `exec` (if successful) never returns!

- starts execution of new program
```c
int main() {
    execl("/bin/echo", "/bin/echo",
          "hello", "world", (void *)0);
    printf("Done exec-ing...
");
    return 0;
}

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

$ gcc execer.c -o execer
$ ./execer
About to exec!
About to exec!
About to exec!
About to exec!
About to exec!
...

```c
int main () {
    if (fork() == 0) {
        exec("/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 lee  staff     8880 Feb 8 01:51 a.out
-rw-r--r--  1 lee  staff      267 Feb 8 01:51 demo.c
Command completed
```
Interesting question:

Why are `fork` & `exec` separate syscalls?

/* i.e., why not: */

fork_and_exec("/bin/ls", ...)

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
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
“handcrafted” process
/etc/inittab

/etc/inittab → init

init

fork & exec

kernel
“Daemons”
e.g., sshd, httpd

kernel

init

fork & exec

fork & exec

getty
kernel

init

shell (e.g. sh)

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`)

**Diagram:**

```
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, " 
");
        argv[i];
        argv[++i] = strtok(NULL, " 
");

    /* fork and run command in child */
    if ((pid = fork()) == 0)
        if (execvp(argv[0], argv) < 0) {
            printf("Command not found\n");
            exit(0);
        }
    
    /* wait for completion in parent */
    waitpid(pid, NULL, 0);
}
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 our processes …

- simultaneously
- with a simple, consistent API
- without 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 holds 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
```c
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  0x000000310acac49d 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
```

**parent**

**child**
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
CPU → \textit{virtual address} → MMU → physical address → Main Memory

MMU:
- address translation unit

- disk address → “swap” space

(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
Virtual address space

$2^n - 1$

Physical address space

$2^{m-1}$

$P_0$

$P_1$

$P_2$
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 Implementations
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 `seg#:offset`
assert (offset ≤ L)
- 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:**

```
0
L
```

**segmentation:**

```
possibly unused!
better!
```
- variable segment sizes $\rightarrow$ 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 *virtual page number & virtual page offset*

- map VPN $\mapsto$ *physical page number*
Given page size = $2^p$ bytes

**VA:**

- virtual page number
- virtual page offset

**PA:**

- physical page number
- physical page offset
VA: virtual page number \rightarrow \text{address translation} \rightarrow \text{physical page number} \rightarrow PA: physical page number

PA: physical page number \rightarrow physical page offset
translation structure: page table

VA: virtual page number virtual page offset

PA: physical page number physical page offset

translation structure: page table

index

valid PPN

2^n entries

if invalid, page is not mapped

if invalid, page is not mapped
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 \(= 1M \times 4 \text{ bytes} = 4MB\)
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*

Diagram:

1. **CPU**
   - **VA:** $N$
   - **Address Translator** (part of MMU)

2. **Page Table walk**
   - **Page Table**

3. **PA:** $N'$

4. **data**

---

- **CPU**
- **Main Memory**
- **Page Table**
- **Address Translator**
- **VA:** $N$
- **PA:** $N'$
- **data**
translation: \textit{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. PA: \(N'\)
9. data
10. data

- CPU
- Address Translator (part of MMU)
- Main Memory
- Page Table
- Kernel
- Disk (swap space)
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 addresses for lookups?
Virtual address based Cache

Process A

Virtual Address Space

Process B

Virtual Address Space

CPU

Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>X</td>
</tr>
<tr>
<td>M</td>
<td>Y</td>
</tr>
<tr>
<td>N</td>
<td>Z</td>
</tr>
</tbody>
</table>

ambiguous!

"synonym" problem
Physical address based Cache

Process A

Virtual Address Space

Process B

Virtual Address Space

CPU

Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>X</td>
</tr>
<tr>
<td>Q</td>
<td>Y</td>
</tr>
<tr>
<td>R</td>
<td>Z</td>
</tr>
</tbody>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
</tr>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>

Physical address based Cache
Q: do caches use physical or virtual addresses for lookups?

A: caches typically use *physical* addresses
saved by hardware:

the *Translation Lookaside Buffer* (TLB) — a cache used solely for VPN→PPN lookups
TLB + Page table
(exercise for reader: revise earlier translation diagrams!)
but!

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

- some architectures store PID (aka “virtual space” ID) in TLB
and 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 # 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!)
Process A

Virtual Memory

Physical Memory

lots of wasted space!

Process B

Virtual Memory
increasing page size results in increased internal fragmentation and lower utilization
i.e., TLB effectiveness needs to be balanced against memory utilization!
so what about 64-bit systems?

\[ 2^{64} = 16 \text{ ExbiByte address space} \approx 4 \text{ billion x 4GB} \]
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 \[= 8 \text{ bytes (64 bits)}\]
- PT size \[= 2^{36} \times 8 = 2^{39} \text{ bytes}\]
  \[= 512\text{GB}\]
512GB

(just for the virtual memory mapping structure)

(and we need one per process)
(these things aren’t going to fit in 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

Page Table

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

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)
PROTECTED-MODE MEMORY MANAGEMENT

The segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a linear address in the processor's linear address space.

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk.

Figure 3-1. Segmentation and Paging

[Diagram showing the relationship between logical address, linear address, segment, page table, page, directory, and physical address space.]
Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels.

Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.
PROTECTED-MODE MEMORY MANAGEMENT

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

Figure 3-2. Flat Model

Figure 3-3. Protected Flat Model

“Flat” model
Table 4-1 illustrates the key differences between the three paging modes. Because they are used only if IA32_EFER.LME = 0, 32-bit paging and PAE paging is used only in legacy protected mode. Because legacy protected mode cannot produce linear addresses larger than 32 bits, 32-bit paging and PAE paging translate 32-bit linear addresses.

Because it is used only if IA32_EFER.LME = 1, IA-32e paging is used only in IA-32e mode. (In fact, it is the use of IA-32e paging that defines IA-32e mode.) IA-32e mode has two sub-modes:

- **Compatibility mode.** This mode uses only 32-bit linear addresses. IA-32e paging treats bits 47:32 of such an address as all 0.
- **64-bit mode.** While this mode produces 64-bit linear addresses, the processor ensures that bits 63:47 of such an address are identical. IA-32e paging does not use bits 63:48 of such addresses.

### Table 4-1. Properties of Different Paging Modes

<table>
<thead>
<tr>
<th>Paging Mode</th>
<th>CR0.PG</th>
<th>CR4.PAE</th>
<th>LME in IA32_EFER</th>
<th>Linear-Address Width</th>
<th>Physical-Address Width¹</th>
<th>Page Size(s)</th>
<th>Supports Execute-Disable?</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>
</tr>
<tr>
<td>32-bit</td>
<td>1</td>
<td>0</td>
<td>0²</td>
<td>32</td>
<td>Up to 40^3</td>
<td>4-KByte 4-MByte</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-KByte 2-MByte</td>
<td>Yes^5</td>
</tr>
<tr>
<td>IA-32e</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>48</td>
<td>Up to 52</td>
<td>4-KByte 2-MByte 1-GByte^6</td>
<td>Yes^5</td>
</tr>
</tbody>
</table>

**Notes:**

1. The physical-address width is always bounded by MAXPHYADDR; see Section 4.1.4.
2. The processor ensures that IA32_EFER.LME must be 0 if CR0.PG = 1 and CR4.PAE = 0.
3. 32-bit paging supports physical-address widths of more than 32 bits only for 4-MByte pages and only if the PSE-36 mechanism is supported; see Section 4.1.4 and Section 4.3.
4. 4-MByte pages are used with 32-bit paging only if CR4.PSE = 1; see Section 4.3.
5. Execute-disable access rights are applied only if IA32_EFER.NXE = 1; see Section 4.6.
6. Not all processors that support IA-32e paging support 1-GByte pages; see Section 4.1.4.

### Paging modes

IIT College of Science

ILLINOIS INSTITUTE OF TECHNOLOGY
IA-32 paging (4KB pages)
IA-32 paging (4MB pages)
IA-32e paging (4KB pages)
The following items describe the IA-32e paging process in more detail as well as how the page size is determined.

- A 4-KByte naturally aligned PML4 table is located at the physical address specified in bits 51:12 of CR3 (see Table 4-12). A PML4 table comprises 512 64-bit entries (PML4Es). A PML4E is selected using the physical address defined as follows:
  - Bits 51:12 are from CR3.
  - Bits 11:3 are bits 47:39 of the linear address.
  - Bits 2:0 are all 0.

  Because a PML4E is identified using bits 47:39 of the linear address, it controls access to a 512-GByte region of the linear-address space.

- A 4-KByte naturally aligned page-directory-pointer table is located at the physical address specified in bits 51:12 of the PML4E (see Table 4-14). A page-directory-pointer table comprises 512 64-bit entries (PDPTEs). A PDPTE is selected using the physical address defined as follows:
  - Bits 51:12 are from the PML4E.

  Figure 4-10. Linear-Address Translation to a 1-GByte Page using IA-32e Paging

IA-32e paging (1GB pages)
Dynamic Memory Allocation

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
from: The Memory Hierarchy
we now have:

Virtual Memory
now what?
- code, global variables, jump tables, etc.
- allocated at fork/exec
- lifetime: *permanent*

**Static Data**
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
- brk pointer marks top of the heap

The Heap
heap mgmt syscall:

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

The Heap
The Heap

\begin{verbatim}
void *hp = sbrk(N);
\end{verbatim}
can use `sbrk` to allocate structures:

```c
int **make_jagged_arr(int nrows, const int *dims) {
    int i, j;
    int **jarr = 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!!!
after the kernel allocates heap space for a process, it is \textit{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

Heap

malloc

sbrk

OS kernel

RAM

Disk

IIT College of Science

ILLINOIS INSTITUTE OF TECHNOLOGY
User Process

application program

dynamic memory allocator

Heap

free(p)

OS kernel

Disk

RAM

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
User Process

application program

dynamic memory allocator

(heap space may not be returned to the kernel!)

free(p)

Heap

OS kernel

RAM

Disk

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
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)