9.7 Case Study: The Intel Core i7/Linux Memory System

We conclude our discussion of virtual memory mechanisms with a case study of a real system: an Intel Core i7 running Linux. Although the underlying Haswell microarchitecture allows for full 64-bit virtual and physical address spaces, the current Core i7 implementations (and those for the foreseeable future) support a 48-bit (256 TB) virtual address space and a 52-bit (4 PB) physical address space, along with a compatibility mode that supports 32-bit (4 GB) virtual and physical address spaces.

Figure 9.21 gives the highlights of the Core i7 memory system. The *processor package* (chip) includes four cores, a large L3 cache shared by all of the cores, and

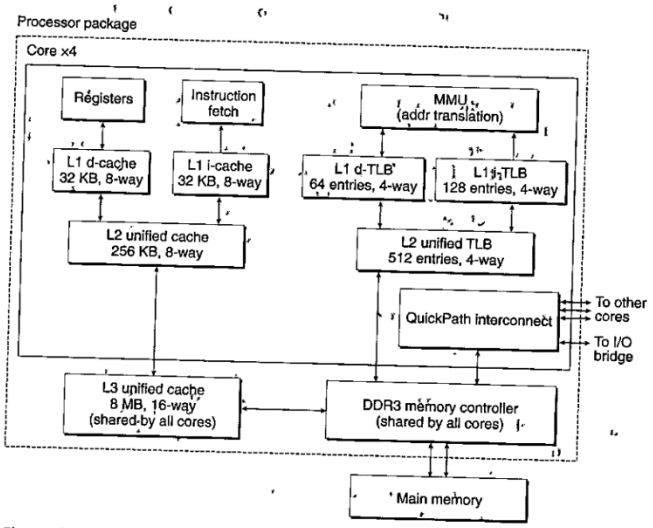


Figure 9.21 The Core i7 memory system.

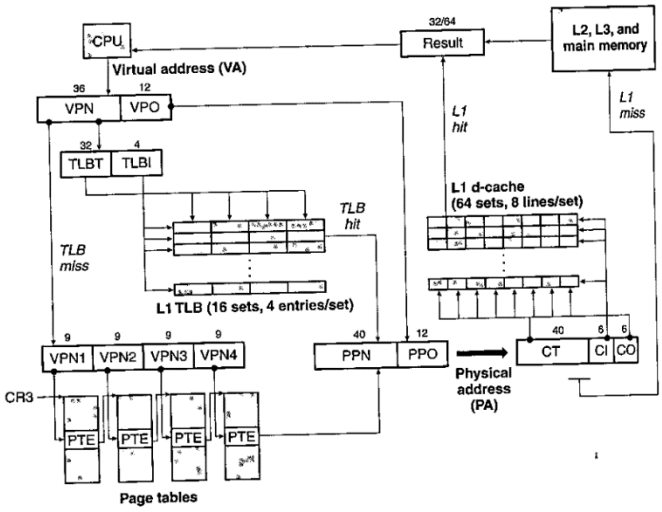
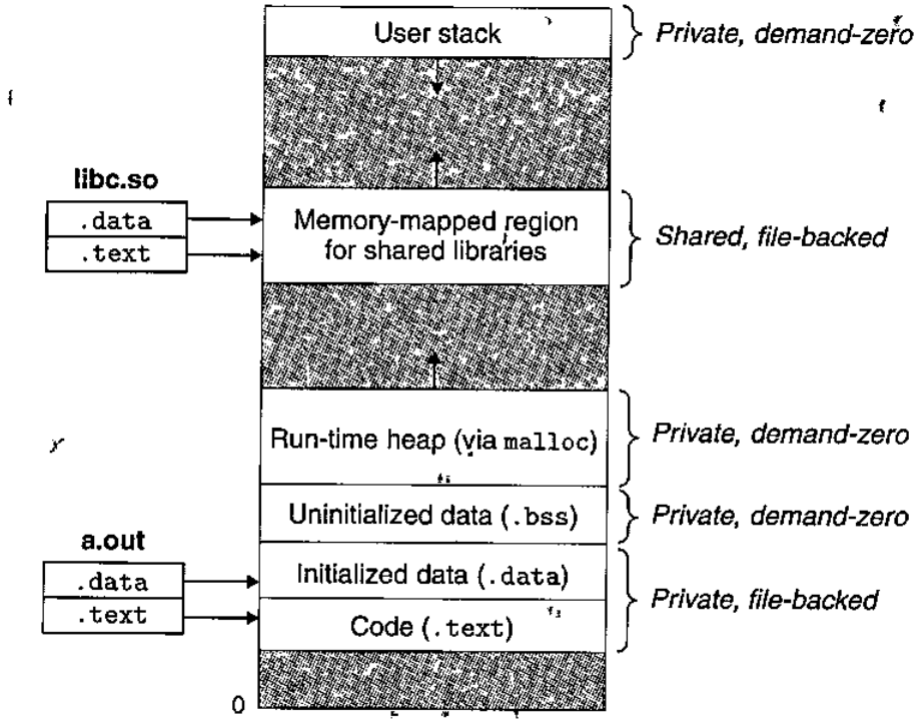
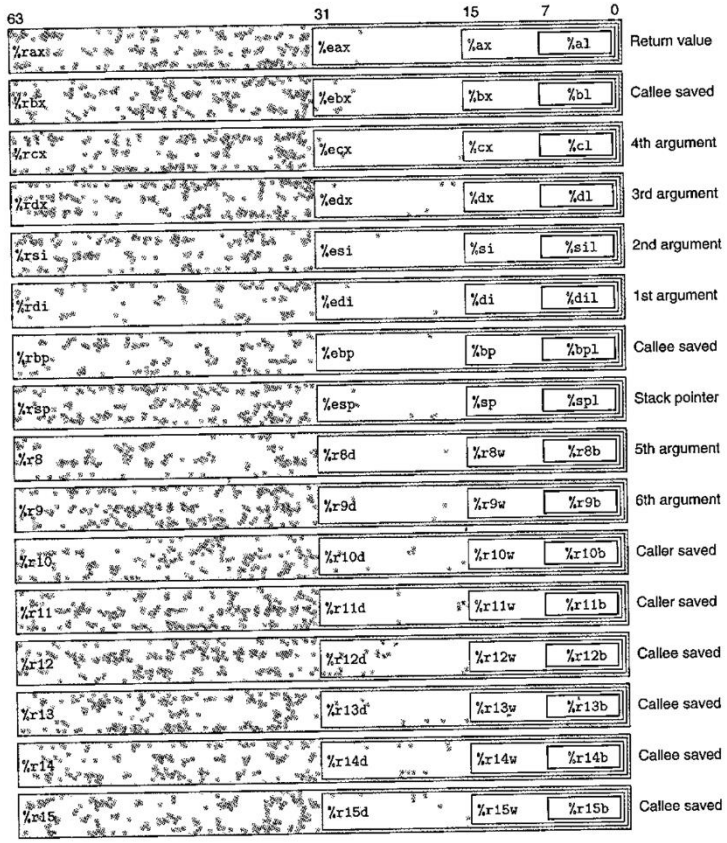


Figure 9.22 **Summary of Core i7 address translation.** For simplicity, the i-caches, i-TLB, and L2 unified TLB are not shown.

a DDR3 memory controller. Each core contains a hierarchy of TLBs, a hierarchy of data and instruction caches, and a set of fast point-to-point links, based on the QuickPath technology, for communicating directly with the other cores and the external I/O bridge. The TLBs are virtually addressed, and 4-way set associative. The L1, L2, and L3 caches are physically addressed, with a block size of 64 bytes. L1 and L2 are 8-way set associative, and L3 is 16-way set associative. The page size can be configured at start-up time as either 4 KB or 4 MB. Linux uses 4 KB pages.

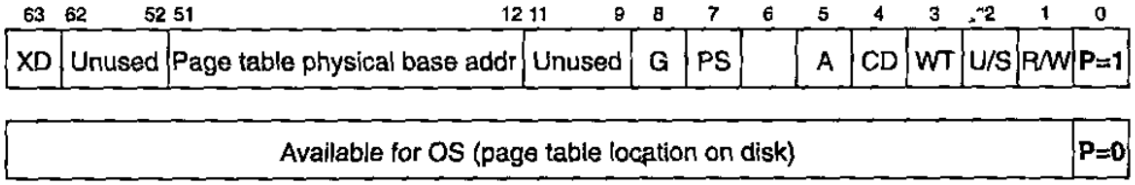

User stack
Private, demand-zero
libc.so
.data
.text
Memory-mapped region for shared libraries
Shared, file-backed
Run-time heap (via malloc)
Private, demand-zero
Uninitialized data (.bss)
Private, demand-zero
a.out
.data
.text
Initialized data (.data)
Code (.text)
Private, file-backed
0


63
31
15
7
0
%rax
%eax
%ax
%al
Return value
%rbx
%ebx
%bx
%bl
Callee saved
%rcx
%ecx
%cx
%cl
4th argument
%rdx
%edx
%dx
%dl
3rd argument
%rsi
%esi
%si
%sil
2nd argument
%rdi
%edi
%di
%dil
1st argument
%rbp
%ebp
%bp
%bpl
Callee saved
%rsp
%esp
%sp
%spl
Stack pointer
%r8
%r8d
%r8w
%r8b
5th argument
%r9
%r9d
%r9w
%r9b
6th argument
%r10
%r10d
%r10w
%r10b
Caller saved
%r11
%r11d
%r11w
%r11b
Caller saved
%r12
%r12d
%r12w
%r12b
Callee saved
%r13
%r13d
%r13w
%r13b
Callee saved
%r14
%r14d
%r14w
%r14b
Callee saved
%r15
%r15d
%r15w
%r15b
Callee saved

```
int NumberOfSetBits(unsigned int i)
{
	i = i - ((i >> 1) & 0x55555555);
	i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
	unsigned split_to_4 = (((i + (i >> 4)) & 0x0F0F0F0F));
	return ( ((split_to_4 >>24)&31) + ((split_to_4 >> 16)&31)
		+ ((split_to_4 >> 8)&31)  + ((split_to_4 )&31) );
}
```

int main(int argc, char* argv[]);

```
#include <sys/mman.h>
void *mmap(void *addr, size_t lengthint " prot ", int " flags ,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```



Field	Description
P	Child page table present in physical memory (1) or not (0).
R/W	Read-only or read-write access permission for all reachable pages.
U/S	User or supervisor (kernel) mode access permission for all reachable pages.
WT	Write-through or write-back cache policy for the child page table.
CD	Caching disabled or enabled for the child page table.
A	Reference bit (set by MMU on reads and writes, cleared by software).
PS	Page size either 4 KB or 4 MB (defined for level 1 PTEs only).
Base addr	40 most significant bits of physical base address of child page table.
XD	Disable or enable instruction fetches from all pages reachable from this PTE.

Figure 9.23 Format of level 1, level 2, and level 3 page table entries. Each entry references a 4 KB child page table.

G = GLOBAL page (don't evict from TLB on task switch)
Level 4 has DIRTY BIT at 6 and PS=0 (PS only defined for L1)
L1,2,3: When P=1 (always for Linux), address field is 40-bit PPN -> beginning of page table in level +1
Level 4: 40-bit PPN points to base of PHYSICAL PAGE. 4KB alignment for physical page and page tables.
CR3 register contains PHYSICAL address of THE L1 page table. CR Control Register

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

#define FILEPATH "/tmp/mmapped.bin"
#define NUMINTS  (1000)
#define FILESIZE (NUMINTS * sizeof(int))

int main(int argc, char *argv[])
{
   int i;
   int fd;
   int result;
   int *map;  /* mmapped array of int's */

   /* Open a file for writing.
    *  - Creating the file if it doesn't exist.
    *  - Truncating it to 0 size if it already exists. (not really needed)
    *
    * Note: "O_WRONLY" mode is not sufficient when mmaping.
    */
   fd = open(FILEPATH, O_RDWR | O_CREAT | O_TRUNC, (mode_t)0600);
   if (fd == -1) {
           perror("Error opening file for writing");
           exit(EXIT_FAILURE);
   }

   /* Stretch the file size to the size of the (mmapped) array of ints
    */
   result = lseek(fd, FILESIZE-1, SEEK_SET);
   if (result == -1) {
           close(fd);
           perror("Error calling lseek() to 'stretch' the file");
           exit(EXIT_FAILURE);
   }

   /* Something needs to be written at the end of the file to
    * have the file actually have the new size.
    * Just writing an empty string at the current file position will do.
    *
    * Note:
    *  - The current position in the file is at the end of the stretched
    *    file due to the call to lseek().
    *  - An empty string is actually a single '\0' character, so a zero-byte
```

```
 *    will be written at the last byte of the file.
 */
result = write(fd, "", 1);
if (result != 1) {
        close(fd);
        perror("Error writing last byte of the file");
        exit(EXIT_FAILURE);
}

/* Now the file is ready to be mmapped.
 */
map = mmap(0, FILESIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (map == MAP_FAILED) {
        close(fd);
        perror("Error mmapping the file");
        exit(EXIT_FAILURE);
}

/* Now write int's to the file as if it were memory (an array of ints).
 */
for (i = 1; i <=NUMINTS; ++i) {
        map[i] = 2 * i;
}

/* Don't forget to free the mmapped memory
 */
if (munmap(map, FILESIZE) == -1) {
        perror("Error un-mmapping the file");
        /* Decide here whether to close(fd) and exit() or not. Depends... */
}

/* Un-mmaping doesn't close the file, so we still need to do that.
 */
close(fd);
return 0;
```

```c
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;

void second(void) {
    printf("second\n");          // prints
    longjmp(buf,1);              // jumps back to where setjmp was called -
making setjmp now return 1
}

void first(void) {
    second();
    printf("first\n");           // does not print
}

int main() {
    if (!setjmp(buf))
        first();                 // when executed, setjmp returned 0
    else                         // when longjmp jumps back, setjmp returns 1
        printf("main\n");        // prints

    return 0;
}
```

ERRNO is a thread local storage (global variable unique to each thread) variable that can be used to give a description of what went wrong.

Sometimes we want it such that when something goes wrong we return back to a specific function. (like try-catch-throw blocks)
int **setjmp**(jmp_buf env) and void **longjmp**(jmp_buf env, int retval) take care of this.

Software Signal Handler (software version of exceptions)
void(***signal**(intsig,void(*func)(int)))(int);

Exceptions are Hardware based and move control to Kernel
-Interrupts: Most commonly from I/O device; **ASYNCHRONOUS**.
§ Trigger interrupt pin -> stop execution -> switch to INTERRUPT HANDLER.
§ This happens within the same thread, and control is handed to kernel.

-Trap: Do things not within the scope of program and need kernels help; **SYNCHRONOUS**
-**Fault**: Caused by potentially recoverable but unexpected errors (like Page Fault); **SYNCHRONOUS**
-Abort: Unrecoverable fatal error such as corrupted memory or hardware error. Exit.
Sync if triggered by MMU (eg. tried to write in read-only); **Async** if RAM breaks or smth.