Systems Programming and Computer Architecture

Gnkgo, Informatik B. Sc. 3. Semester

Contents

1 Calculations **2**
1.1 Nested Array Row . . . 2
1.2 IEEE . . . 2
1.3 Cache . . . 2

2 Parallel Programming **2**
2.1 Solving ABA problem . . . 2

3 Architecture **2**
3.1 Memory . . . 3
3.2 Locks . . . 3
3.3 Jumps . . . 4
3.4 MESI . . . 5
3.5 Program Order . . . 5
3.6 Visibility Order . . . 5
3.7 Sequential Consistency . . . 5

4 Dependence **5**
4.1 Order on stack . . . 5

5 Exceptions **5**
5.1 Abort . . . 6
5.2 Fault . . . 6
5.3 Trap . . . 6

6 Linker **6**

7 Toolchain **6**

8 Debugger **6**

9 Random **7**

1 Calculations

1.1 Nested Array Row

`arr[N][M]`

`arr[i][j] = arr + (i * M + j)`

1.2 IEEE

Exp = 000 ... 0 → -Bias + 1, leading 0.

1.3 Cache

Each page table entry PTE contains only the PPN and the metadata

With multilevel decoding, from virtual address to physical address virtual address space - page offset = bits to be encoded. Bits to be encoded / levels

Minimum size of each PTE in bytes Physical address space - Pagesize + Flags(Metadata)

Size of processor's page table Virtual address space - pagesize + PTE size

False Sharing Poor performance when on different processors → flush whole cache even though they didn't need the same location. Good performance when on the same processor

Number of misses Two arrays map to the same cache line if the address % cachesize is the same. Then you would have 100% cache misses

2 Parallel Programming

2.1 Solving ABA problem

- **Hazard Pointers:** Threads keep track of questioned pointers in a shared data structure. Each thread knows that the object on the given memory address defined by the pointer might have been modified by another thread.
- **Immutability:** The usage of immutable objects solves this problem, as we don't reuse objects across the application.
- **Double Compare and Swap:** Keep track of one more variable, which is the version number.

3 Architecture

Section header table Offsets and sizes of each section

symtab. Symbol table, procedure and static variable names, section names and locations

.rel.text Relocation info for .text section instructions for modifying

.debug* Info for symbolic debugging (gcc. -g)

It exists 1 virtual address space per process

Metadata bits in PTE

- **D:** Dirty
- **A:** Accessed
- **P:** Page is present in physical memory (1) or not(0)
- **Avail:** Available for system programmers

3.1 Memory

Performance is *not uniform*! Cache and virtual memory effects can greatly affect program performance.
Typed Different kinds of memory behave differently.
Not unbounded It must be allocated and managed.
Non-Uniform Memory Access (NUMA) Removes bottleneck

- Multiple, independent memory banks
- Processors have independent paths to memory
- Cannot snoop on the bus anymore → it's not a bus! Use a message-passing interconnect

Solution1: Bus emulation

- Similar to snooping but without a shared bus
- Each node sends a message to all other nodes (e.g., read exclusive)
- Waits for a reply from all nodes before proceeding (e.g., acknowledge)
- *AMD coherent HyperTransport*

Solution2: Cache Directory

- Augment each node's local memory with a cache directory

Directory-based Cache Coherence

- Home node maintains a set of nodes that may have line Large multiprocessors
- More efficient when:
  - Lines are not widely shared
  - Lots of NUMA nodes
  - Avoid broadcast/incast
  - Reduces interconnect traffic, load at each node
  - Requires lots of fast memory

3.2 Locks

MCS locks

- *Problem:* Cache line containing lock is a *hot spot*
  - Continuously invalidated as every processor tries to acquire it
  - Dominates interconnect traffic
- *Solution:* When acquiring, a processor enqueues itself on a list of waiting processors, and spins on its *own entry* in the list

```
struct qnode {
    struct qnode *next;
    int locked;
};
typedef struct qnode *lock_t;

void acquire(lock_t *lock, struct qnode *local) {
    local -> next = NULL;
    struct qnode *prev = XCHG(lock, local);
    if (prev) { // queue was non-empty
        local-> locked = 1;
        prev -> next = local;
```

```
        while (local -> locked); // spin
    }
}

void release(lock_t *lock, struct qnode *local) {
    if (local->next == NULL) {
        if (CAS(lock, local, NULL)) {
            return;
        }
        while(local->next ==NULL); // spin
    }
    local -> next -> locked = 0;
}
```

Flags

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Non-negative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Figure 1: Flags Table

Synonym

- Only in virtual part
- Different VAs map to the same PA
- VV with keys remove homonyms but create synonyms

Homonym

- Same name for different data
- Same VA refers to different PAs
- Flush cache on context switch
- Force non-overlapping address-space
- Tag VA with address-space

3.3 Jumps

Longjmp: If retval = 0, longjmp returns 1
Setjump: Saves the current calling environment

- Program counter (not necessary)
- Stack pointer
- General-purpose register

Or as it was mentioned in the exam:

- Current stack pointer
- The current program counter (%rip)
- Callee-save processor registers
- Frame pointer register §rbp

3.4 MESI

- Dirty data always written through memory
- No cache cache transfer
- Good if latency of memory << latency of remote cache

3.5 Program Order

Order in which a program on a processor appears to issue reads and writes. Refers only to **local** reads/writes

3.6 Visibility Order

Order in which all reads and writes are seen by one or more processors. Refers to all operations in the machine

3.7 Sequential Consistency

Think of PPROG → Order is important but you can move instructions horizontally

1. Operations from a processor appear (to all others) in program order
2. Every processor's visibility order is the interleaving of all the program orders

Requirements:

- Each processor issues memory ops in program order
- Memory operations are atomic

4 Dependence

Anti-dependence write after read, can avoid hazard with register renaming
Output-dependence write after write, can avoid hazard with register renaming
True-dependence read after write

4.1 Order on stack

Extra args
Return address
Saved base pointer
Local variables

5 Exceptions

Processor Exception changes the control flow exceptionally and most of the time context switches to the OS for further instructions on how the given exception should be handled. Exceptions are associated with an exception code that at the same time is an index into an exception table defined by the OS. In contrast to, for example, Java exceptions, processor exceptions are not always bad; a system call is often handled as an exception too.
Asynchronous exception caused by events outside of the processor → ctrl-c (interrupt)
Synchronous exception caused by an instruction of the processor

5.1 Abort

- Nonrecoverable error
- Sync
- Unintentional
- Machine check
- **Aborts current program**

5.2 Fault

- Potentially recoverable error
- Sync
- Unintentional
- Page fault, protection fault
- **Either re-executes the faulting instruction or abort**

5.3 Trap

- Intentional
- Sync
- System calls, breakpoint trap
- **Returns control to "next" instruction**

6 Linker

- Resolving symbol references from other files
- Merging different object file sections

7 Toolchain

How C code turns into an executable file
C source - Preprocessor - Compiler - Assembler - Linker - Executable
GNU gcc Toolchain
CPP (Macro substitution include header files) - CC1 (compile each C file into assembly language) - as (Assemble each file into object code) - ld (link object files into program binary) - Executable

8 Debugger

Valgrind helpful for memory-related errors
GDB conventional debugger, good for finding bad pointer dereferences, hard to detect the other memory bugs
Objdump Useful tool for examining object code, analyzes the bit pattern of series of instructions

9 Random

\\is an escape.

PIC, Programmable Interrupt Controller

- Map physical interrupt pins to interrupt vectors
- Buffer simultaneous interrupts → Won't lose some device's interrupt
- Prioritize interrupts
- Selectively mask any individual device's interrupts useful for high interrupt rate

How many BxB blocks fit into a cache and how many blocks do you need? → $XB^2 < C$ where X is mostly 3 and C the Cache size

Reading a string of length 10 in the buffer

```
fgets(buf, 10, stdin);
fscanf(stdin, "%9s", buf);

NULL = (void *) 0

strncpy(char *dest, const char *source, size)
```

DRAM as cache: Fully associative, sophisticated replacement policy, writeback

```
void print_integers(unsigned num_ints, char *msg, ...) {
    va_list ap; // Variable which keeps track of which argument we're currently looking at, a bit like an
    va_start(ap, msg); // Initialize the iterator based on the last fixed argument;
    for (int i = 0; i < num_ints; i++>) {
        int j = va_arg(ap, int);//return the next argument, cast to an int
        printf("int %d = %d\n", i, j);
    }
    va_end(ap) //free up the iterator
}
```