Efficiency: locality; OS intro: Multiprocessing: limited direct execution; system calls; processes

COSC 208, Introduction to Computer Systems, 2022-04-14

Announcements

- Project 3 due Monday

Outline

- Locality
- Optimizing loops for locality
- Limited direct execution
- System calls
- Process abstraction
- Creating processes

Temporal vs. spatial locality

- *What is temporal locality?*
    - Access the same data repeatedly
    - E.g., for loop variable
- *What is spatial locality?*
    - Access data with a similar scope
    - E.g., next item in array
    - E.g., local variables/parameters, which are stored in the same stack frame
- Analogies for temporal and spatial locality
    - Book storage (Dive Into Systems Section 11.3.2)
        - Temporal locality — store most frequently used books at your desk, less frequently used books on your bookshelf, and least frequently used books at the library
        - Spatial locality — checkout books on the same/nearby subjects when you go to the library
    - Groceries
        - Temporal locality — you store food you eat frequently in the front of the refrigerator, while you store food you eat infrequently in the back of the refrigerator
        - Spatial locality — you organize the items on your grocery list based on the aisle in which they are located
    - Breakout groups: *Develop your own analogy for temporal and spatial locality*

Optimizing loops for locality

- Techniques
    - Loop interchange — with nested loops, swap inner and outer loop
    - Loop fusion — combine two loops at the same level into a single loop
    - Loop fission — split a single loop into two loops at the same level

- Q1: *Modify the following function to improve spatial locality*

```
int *hundreds() {
    int *nums = malloc(sizeof(int) * 1000);
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 1000; j+= 100) {
            nums[i+j] = i;
        }
    }
}
```

Perform loop interchange

```
int *hundreds_optimized() {
    int *nums = malloc(sizeof(int) * 1000);
    for (int j = 0; j < 1000; j+= 100) {
        for (int i = 0; i < 100; i++) {
            nums[i+j] = i;
        }
    }
}
```

- Q2: *Modify the following function to improve temporal locality*

```
int odds(int *nums, int length) {
    for (int i = 0; i < length; i++) {
        nums[i] = nums[i] % 2;
    }
    int count = 0;
    for (int j = 0; j < length; j++) {
        count += nums[j];
    }
    return count;
}
```

Perform loop fusion

```
int odds_optimized(int *nums, int length) {
    int count = =0;
    for (int i = 0; i < length; i++) {
        nums[i] = nums[i] % 2;
        count += nums[i];
    }
    return count;
}
```

- Q3: *Modify the following function to improve spatial locality*

```
void multiplication(int grid[][], int rows, int cols) {
    for (int c = 0; c < cols; c++) {
        for (int r = 0; r < rows; r++) {
            grid[r][c] = c * r;
        }
    }
}
```

Perform loop interchange

```
void multiplication_optimized(int grid[][], int rows, int cols) {
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            grid[r][c] = c * r;
        }
    }
}
```

- Q4: *Modify the following function to improve temporal locality*

```
long stdev(int *nums, int length) {
    long sum = 0;
    for (int i = 0; i < length; i++) {
        sum += nums[i];
    }
    int mean = sum / length;
    sum = 0;
    for (int j = 0; j < length; j++) {
        int diff = nums[j] - mean;
        sum += diff * diff:
    }
    mean = sum / length;
    return sqrt(mean);
}
```

It's not possible to optimize this code more—there are no loops to interchange; there are no loops to fission; the loops cannot be fused.

Operating systems (OS) overview

- Purpose of an OS
  - Make computer hardware easy to use—e.g., an OS knows how to load an application's executable code from persistent storage (e.g., solid state drive (SSD)) into main memory, initialize the process's address space (code, heap, stack), and make the CPU execute the application's instructions
  - Support multiprocessing—i.e., running multiple applications concurrently
    - Concurrently == switch between multiple tasks during a window of time—e.g., alternate between cooking and setting the table
    - Simultaneously == complete two tasks at the same time—e.g., listen to a podcast while cooking
  - Allocate and manage hardware resources—e.g., decide when/which applications can use the CPU, decide when/which memory applications can use, prevent applications from stealing/accessing another application's CPU time or memory
  - Many OSes also provide a user interface (UI)
- How does the OS fulfill its duties?
  - Mechanisms — fundamental approaches for managing/providing access to hardware resources
    - E.g., system calls, process abstraction
  - Policy — specific ways of employing an approach; different policies make different trade-offs (in terms of efficiency, performance, etc.)
    - E.g., process scheduler

Accessing hardware

- OS is responsible for allocating/managing the hardware
  - ⇒ applications should **not have unfettered access to hardware**
- *How should applications access the hardware?*
  - Ask the OS for access to the hardware
    - How do we ensure the OS does not "lose control" of the hardware?
  - Asks the OS to perform an action on the application's behalf
    - How do we ensure this doesn't substantially degrade performance?
- Example: execute an instruction on the CPU
  - Asking the OS to do this on behalf of an application is impractical—OS would need to execute multiple assembly instructions for each assembly instruction the application wants to execute
  - Alternative: allow the application to execute its own instructions on the CPU
    - This is risky—an application may execute an instruction that controls the hardware, e.g., `hlt` (halt) instruction pauses the CPU
  - Alternative: allow the application to execute "safe" instructions on its own on the CPU
- Example: accessing the solid state drive (SSD)
  - Allowing the application to access the SSD directly
    - This is risky—an application may read/write data it should not be able to access
  - Alternative: asking the OS to access the SSD on the application's behalf
    - Latency of accessing SSD (~1 million CPU cycles) far outweighs the extra instructions required for OS to perform the access on the application's behalf
- Mechanisms
  - Limited Direct Execution (LDE)
  - System calls

Limited Direct Execution

- CPU has two modes of execution: user mode & kernel mode
- *When does a CPU run in user mode?* — when executing application code
- *When does a CPU run in kernel mode?* — when executing OS code
- Allowable operations in user mode are restricted
    - Applications can...
        - Perform arithmetic/logic operations
        - Load/store values in its stack/heap
    - Applications must ask the OS to...
        - Start/terminate applications
        - Create/delete files
        - Display output on screen
        - Read input from user
    - Must transfer control to the OS to perform these operations — How?

System calls

- Invoked via a special assembly instruction: trap (generic) or `svc` (on ARM)
    - Example program

```c
#include <stdio.h>
#include <unistd.h>
int user() {
    int uid = getuid();
    return uid;
}
int main() {
    int u = user();
    printf("User %d is running this process\n", u);
}
```

    - Assembly code

```
00000000004006ac <user>:
    4006ac: d10083ff    sub sp, sp, #0x20
    4006b0: f9000bfe    str x30, [sp, #16]
    4006b4: 94007713    bl  41e300 <__getuid>
    4006b8: b9000fe0    str w0, [sp, #12]
    4006bc: b9400fe0    ldr w0, [sp, #12]
    4006c0: f9400bfe    ldr x30, [sp, #16]
    4006c4: 910083ff    add sp, sp, #0x20
    4006c8: d65f03c0    ret
000000000041e300 <__getuid>:
    41e300: d28015c8    mov x8, #0xae
    41e304: d4000001    svc #0x0
    41e308: d65f03c0    ret
```

    - Functions in the C standard library that involve a privileged operation (e.g., `printf`) put the system call number in a register and invoke a trap instruction — programmer doesn't have to worry about these details; they can just call the appropriate function in the C standard library

- When `svc` is executed
    1. CPU saves registers to the kernel stack — kernel stack is at a fixed location in memory
        - *Why do we need to save the registers?* — so we can return to `user` when `__getuid` is done
    2. CPU switches to kernel mode
    3. CPU uses system call number to index into table of trap handlers
        - At boot, initialize table of trap handlers with pointers into OS code for handling each type of syscall
    4. Branch and link to trap handler code
    5. CPU restores registers from the kernel stack
    6. CPU switches to user mode
    7. Resume execution after `svc`
- *What should we do if a process tries to perform a privileged operation without making a system call?*
    - Let the code keep running — code may assume privileged operation was successful
    - Kill the process

Process abstraction

- Process — running program and its corresponding machine state (code, memory, and register values)
    - Program is static code and static data; process is dynamic instance of code and data
    - Cooking analogy
        - Recipe == program
        - Following a recipe == process
        - Ingredients == program inputs
        - Prepared food == program outputs
        - Cabinets, fridge, etc. == persistent storage
        - Prep area (e.g., counter) == registers & main memory
        - Contents and status of the prep area; current step of recipe == machine state
        - Chef == CPU
    - Can have multiple processes all running different instances of the same program
        - Cooking analogy — chef may be making multiple batches of the same recipe
- OS is responsible for…
    - Creating processes — when a user or another application requests it
    - Scheduling processes — i.e., deciding when/which process should be allowed to use the CPU
    - Switching processes — i.e., saving the machine state of one process and restoring the machine state of another process; called context switching
    - Cleaning-up processes — i.e., releasing any resources they are using when the process is done
    - Interacting with I/O devices (e.g., disks, NICs, graphics card) on behalf of processes
- Q1: *Consider building a Lego kit as an analogy for operating systems' process abstraction. Match each component of the analogy with the corresponding component of a real computer system.*
    - Instruction booklet == program
    - Following the assembly instructions == process
    - Lego bricks == program inputs
    - Completed kit == program outputs
    - Cabinet/drawers for storing Legos == persistent storage
    - Building area (e.g., tabletop) == registers and main memory
    - Current step for the instruction booklet == program counter
    - You == CPU

Creating processes

- `int fork()`
  - Creates an exact copy of the running process, except for the return value from `fork` — return `0` to child (i.e., new) process; return child's process ID to parent process (i.e., process that called fork)
  - Both child and parent resume execution from place where `fork` was called
- Q2: *What does the following code output?*

```c
int main(int argc, char **argv) {
    printf("Before fork\n");
    int pid = fork();
    printf("After fork\n");
    return 0;
}
```

```
Before fork
After fork
After fork
```

- Q3: *What does the following code output (assuming the new process has PID 1819)?*

```c
int main(int argc, char **argv) {
    printf("Before fork");
    int pid = fork();
    if (pid == 0) {
        printf("Child gets %d\n", pid);
    } else {
        printf("Parent gets %d\n", pid);
    }
    return 0;
}
```

```
Before fork
Child gets 0
Parent gets 1819
```

OR

```
Before fork
Parent gets 1819
Child gets 0
```

- Creating an exponentially increasing number of processes (known as a *fork bomb*)

```c
int main() {
    while(1) {
        fork();
```

```
    }
}
```