Boring Stuff

- Shell Lab due next Tuesday, 9 July 2013
  - Your first concurrency assignment!
  - If you like process level concurrency, take OS 😊
- Malloc Lab out the same time Shell Lab is due
  - My favorite lab!
  - Design and implement a memory allocator
- Pressing concerns?
Menu for Today

- Advice, not Motivation
- The Rest of Shell Lab
  - I/O (with Pictures!)
  - How to sigsuspend()
  - Minor Details
- Virtual Memory
- Address Translation
- Extra: C Primer
Rubber Duck Debugging

“To use this process, a programmer explains code to an inanimate object, such as a rubber duck, with the expectation that upon reaching a piece of incorrect code and trying to explain it, the programmer will notice the error.”
MORE ON DUCKS
I/O

- Four basic operations
  - open()
  - close()
  - read()
  - write()

- What’s a file descriptor?
  - Returned by `open()`
  - Some positive value, or -1 to denote error
  - `int fd = open("/path/to/file", O_RDONLY);`
FILE DESCRIPTORS

- Every process starts with these 3 by default
  - 0 – STDIN
  - 1 – STDOUT
  - 2 – STDERR

- Every process gets its own file descriptor table
  - Used to refer to the opened files

- Forked processes share open file tables
**Parent and Child After fork()**

- Shamelessly stolen from lecture:

  **Descriptor table**
  [one table per process]

  **Open file table**
  [shared by all processes]

  ![Diagram showing parent and child descriptors and open file tables](image-url)
What is dup2()?

- Copies file descriptor entries
  - Causes the entries to point to the same files as another file descriptor
- Takes the form: `dup2(dest_fd, src_fd)`
  - src_fd will now point to the same place as dest_fd
**DUP2() SUPER RELEVANT: BEFORE**

- **Goal:** Redirect stdout
- **First, use** `open()` **to open a file to redirect**
  - **For Shell Lab:** Done right before the call to `exec()` in the child process
  - This example, fd 4 is the file descriptor of the opened file
**DUP2() SUPER RELEVANT: AFTER**

- To redirect, duplicate fd 4 into fd 1.
- Call `dup2(4, 1)`
  - Causes fd 1 to refer to disk file pointed at by fd 4
- Accessing fd 1 will now get you File B
SIGSUSPEND() BACKGROUND

What is sigsuspend()?
- Used to protect critical regions from signal interruption.
- It is especially useful for (you guessed it) “pausing” or “sleeping” while waiting for a signal.
- Much better solution to the “sleep loop”

Goal: to block all the way up until the instruction our process is suspended.
SIGSUSPEND() DETAILS

- int sigsuspend(const sigset_t *sigmask);
  - Where *sigmask contains a mask of signals YOU DON’T want to be interrupted by
  - Can be considered opposite of sigprocmask() which takes a mask of signals you want to operate on.

- Quick example: if you want to be woken up from sigsuspend() by SIGCHLD, it better not be in the mask you pass in!
**How to sigsuspend()**

```c
int main() {
    sigset_t waitmask, newmask, oldmask;

    /* set with everything except SIGINT */
    sigfillset(&waitmask);
    sigdelset(&waitmask, SIGINT);

    /* set with only SIGINT */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /* oldmask contains the mask of signals before the 
     * block with newmask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        unix_error("SIG_BLOCK error");

    /* "CRITICAL REGION OF CODE" - (SIGINT blocked) */

    /* Pause, allowing ONLY SIGINT */
    if (sigsuspend(&waitmask) != -1)
        unix_error("sigsuspend error");

    /* RETURN FROM SIGSUSPEND -- (Returns to signal 
     * state from before sigsuspend) */
    /* Reset signal mask which unblocks SIGINT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        unix_error("SIG_SETMASK error");
}
```

**Points of interest**

- **Sigprocmask()** fills oldmask with the signal mask from before SIG_BLOCK
- If sigsuspend() returns from being awoken, it returns 1.
- After sigsuspend() returns, the state of the signals returns to how it was before the call
Questions to ponder:

- How can we grow processes safely?
- What to do about fragmentation?
- How can we make large contiguous chunks fit easier?
HOW DO WE SOLVE THESE PROBLEMS?

- We are scientists (and engineers)...
  - Insert a level of indirection
VIRTUAL MEMORY

- Is the Best Thing Ever™
  - Demand paging
  - Memory Management
  - Protection

- Allows the *illusion of infinite memory*
  - Kernel manages page faults

- Each process gets its own virtual address space
  - Mapping is the heart of virtual memory
**VM of a Linux Process**

**Different for each process**
- Process-specific data structs (ptables, task and mm structs, kernel stack)
- Physical memory
- Kernel code and data
- User stack
- Memory mapped region for shared libraries

**Identical for each process**
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

**Kernel virtual memory**
- Process virtual memory

Process virtual memory

- Process virtual memory

- Physical memory

- Kernel code and data

- User stack

- Memory mapped region for shared libraries

- Runtime heap (malloc)

- Uninitialized data (.bss)

- Initialized data (.data)

- Program text (.text)

Process virtual memory

- Process virtual memory

- Physical memory

- Kernel code and data

- User stack

- Memory mapped region for shared libraries

- Runtime heap (malloc)

- Uninitialized data (.bss)

- Initialized data (.data)

- Program text (.text)
Virtual address

Virtual page number (VPN)  Virtual page offset (VPO)

Page table address for process

Valid bit = 0: page not in memory (page fault)

Page table base register (PTBR)

Page table

Physical page number (PPN)

Physical page offset (PPO)
Overview of a Hit

CPU Chip

CPU

1. VA

MMU

2. PTEA
3. PTE
4. PA

Cache/Memory

Data

5
### Two-Level Page Table

**Level 1 page table**

- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- (1K - 9) null PTEs

**Level 2 page tables**

- PTE 0
- ... PTE 1023

**Virtual memory**

- VP 0
- ... VP 1023
- VP 1024
- ... VP 2047
- Gap
- 1023 unallocated pages
- VP 9215
- ...

---

32 bit addresses, 4KB pages, 4-byte PTEs
TRANSFIGURING W/ A K-LEVEL PAGE TABLE

VIRTUAL ADDRESS

PHYSICAL ADDRESS
**But Memory Accesses are Slow**

- At least 2 memory accesses
  - Fetch page-table entry (PTE) from memory
  - Then fetch data from memory
- In x86, 3 memory accesses
  - Page directory, page table, physical memory
- In x86_64, 4 level page-mapping system
- What should we do?
  - Please don’t say insert a level of “indirection”
TRANSLATION LOOKASIDE BUFFER (TLB)

- Super fast hardware cache of PTEs
- Idea: Locality exists between memory accesses
  - Typically access nearby memory
  - Usually on the same page as current data
    - Arrays with loops
    - Program instructions
VM: TRANSLATIONS w/ TLB AND TABLES

Virtual address

Virtual page number (VPN)  Virtual page offset (VPO)

Physical address

Physical page number (PPN)  Physical page offset (PPO)

Translation Lookaside Buffer (TLB)

Page table base register (PTBR)

Page table address for process

Translation Lookaside Buffer (TLB)

Virtual page number (VPN)  Virtual page offset (VPO)

Page table

Valid  Physical page number (PPN)

TLB Hit: Fetch straight from TLB

TLB Miss: Do a page walk to fetch the entry

Physical page number (PPN)  Physical page offset (PPO)
Overview of a TLB Hit

CPU Chip

CPU

MMU

TLB

VPN

PTE

PA

Cache/Memory

Data

1. VA

2. VPN

3. PTE

4. PA

5. Data
Overview of a TLB Miss

CPU Chip

1. CPU
2. TLB
3. PTEA
4. PTE
5. PA
6. Data

MMU

Cache/Memory

VPN
TUTORIAL: VIRTUAL ADDRESS TRANSLATION

- **Addressing**
  - 32 bit virtual address
  - 32 bit physical address
  - Page size = 4 kb

- **Paging**
  - 10 bit page directory index
  - 10 bit page table index
  - 12 bit offset

- **TLB**
  - Direct Mapped
  - 4 entries
**Tutorial: Address Translation Hit**

Always access TLB first
**Tutorial: Address Translation Hit**

Set Tag PPN Valid
---
0 3A7AE 5F7F7 1

VPO and PPO are always the same!

Physical Page Number

Physical Page Offset
TUTORIAL: ADDRESS TRANSLATION MISS

TLBT  TLBI  VPO

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3B8AC</td>
<td>DEAD</td>
<td>0</td>
</tr>
</tbody>
</table>

- TLB Miss! Do page walk
TUTORIAL: ADDRESS TRANSLATION MISS

<table>
<thead>
<tr>
<th>PDI</th>
<th>PTI</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>3B8</td>
<td>2B3</td>
<td>BEE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Directory Index</th>
<th>Page Table Address</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x3B8</td>
<td>0xFAFF8034</td>
<td>1</td>
</tr>
<tr>
<td>0x3B9</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
# Tutorial: Address Translation Miss

## Table

<table>
<thead>
<tr>
<th>PDI</th>
<th>PTA</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>0x3B8</td>
<td>0xFAFF8034</td>
<td>1</td>
</tr>
<tr>
<td>0x3B9</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

## Diagram

- **3B8**: PDI
- **2B3**: PTI
- **BEE**: VPO

### Address Translation Miss

- **0xFAFF8034**
  - PTI
  - PPN
  - Valid
  - ..
  - ...
  - ...
  - ...

- **0xFAFF9034**
  - PTI
  - PPN
  - Valid
  - ..
  - ...
  - ...
  - ...
  - ...
  - ..
## Tutorial: Address Translation Miss

<table>
<thead>
<tr>
<th>PDI</th>
<th>PTI</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>3B8</td>
<td>2B3</td>
<td>BEE</td>
</tr>
</tbody>
</table>

### Page Table Index

<table>
<thead>
<tr>
<th>Page Table Index</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>..</td>
<td>..</td>
<td>...</td>
</tr>
<tr>
<td>0x2B2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x2B3</td>
<td>0x2D00D</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
TUTORIAL: ADDRESS TRANSLATION MISS

Physical Page Number

Physical Page Offset
TRANSLATION MACRO EXERCISE

- 32 bit address: 10 bit VPN1, 10 bit VPN2, 12 bit VPO
- 4KB pages
- Define the following function like macros:
  - Page align
    
    ```
    #define PAGE_ALIGN(v_addr) _________________________________
    ```
  - Gets VPN1/VPN2 as unsigned int from virtual address
    ```
    #define VPN1(v_addr) _______________________________________
    #define VPN2(v_addr) _______________________________________
    ```
  - Gets VPO as unsigned int from virtual address
    ```
    #define VPO(v_addr) ________________________________________
    ```
  - Calculates the address of the page directory index
    ```
    #define PDEA(pd_addr, v_addr) ______________________________
    ```
  - Calculate address of page table entry
    ```
    #define PTEA(pd_addr, v_addr)_______________________________
    ```
  - Calculate physical address
    ```
    #define PA(pd_addr, v_addr) ________________________________
    ```
TRANSLATION MACRO SOLUTION

- 32 bit address: 10 bit VPN1, 10 bit VPN2, 12 bit VPO
- 4KB pages
- Define the following function like macros:
  - Page align
    ```
    #define PAGE_ALIGN(v_addr) (((unsigned int) v_addr & ~0xffff)
    ```
  - Gets VPN1/VPN2 as unsigned int from virtual address
    ```
    #define VPN1(v_addr) (((unsigned int) (((v_addr)>>22)&0x3ff))
    #define VPN2(v_addr) (((unsigned int) (((v_addr)>>12)&0x3ff))
    ```
  - Gets VPO as unsigned int from virtual address
    ```
    #define VPO(v_addr) (((unsigned int) (((v_addr)&0xffff))
    ```
  - Calculates the address of the page directory index
    ```
    #define PDEA(pd_addr, v_addr) (((void **)pd_addr)+VPN1(v_addr))
    ```
  - Calculate address of page table entry
    ```
    #define PTEA(pd_addr, v_addr) 
      (((void **)PAGE_ALIGN(*PDEA(pd_addr,v_addr)))+VPN2(v_addr))
    ```
  - Calculate physical address
    ```
    #define PA(pd_addr, v_addr) 
      (((PAGE_ALIGN(*PTEA(pd_addr,v_addr)))) | VPO(v_addr))
    ```
Extra Stuff

- For next week, or for your enjoyment
All the C!

- “Saving you from malloc misery…”
- Basics
- Useful C Stuff
- Debugging
C AND POINTER BASICS

- Statically allocated arrays:
  - int prices[100];
  - Get rid of magic numbers:
    - int prices[NUMITEMS];

- Dynamically allocated arrays:
  - int *prices2 = (int *) malloc(sizeof(int) * var);

- Which is valid:
  - prices2 = prices;
  - prices = prices2;

- The & operator:
  - &prices[1] is the same as prices+1

- Function Pointer:
  - int (*fun)();
  - Pointer to function returning int
**Peeling the Onion (K&R p.101)**

- `char **argv`  
  - argv: pointer to a pointer to a char

- `int (*daytab)[13]`  
  - daytab: pointer to array[13] of int

- `int *daytab[13]`  
  - daytab: array[13] of pointer to int

- `char (((x()))[][])()`  
  - x: function returning pointer to array[] of pointer to function returning char

- `char (*(*x[3])())[5]`  

- **Takeaway**  
  - There is an algorithm to decode this (see K&R p. 101)  
  - Always use parenthesis!!
Why Typedefs?

- For convenience and readable code
- Example:
  ```c
  typedef struct {
      int x;
      int y;
  } point;
  ```
- Function Pointer example:
  ```c
  typedef int(*pt2Func)(int, int);
  pt2Func is a pointer to a function that takes 2 int arguments and returns an int
MACROS ARE COOL

- C Preprocessor looks at macros during the preprocessing step of compilation
- Use `#define` to avoid magic numbers:
  - `#define TRIALS 100`
- Function like macros – short and heavily used code snippets
  - `#define GET_BYTE.ONE(x) ((x) & 0xff)`
  - `#define GET_BYTE.TWO(x) (( (x) >> 8) & 0xff)`
- Also look at inline functions (example prototype):
  - `inline int max(int a, int b)`
  - Requests compiler to insert assembly of `max` wherever a call to `max` is made
- Both useful for malloc lab
DEBUGGING – FAVORITE METHODS

- Using the `DEBUG` flag:
  - `#define DEBUG`
    ```
    #ifdef DEBUG
    #define dbg_printf(...) printf(__VA_ARGS__)
    #else
    #define dbg_printf(...) 
    #endif
    ```

- Compiling (if you want to debug):
  - `gcc -DDEBUG foo.c -o foo`

- Using `assert`
  - `assert(posvar > 0);`
  - `man 3 assert`

- Compiling (if you want to turn off asserts):
  - `gcc -DNDEBUG foo.c -o foo`
LITTLE THINGS

- Usage messages
  - Putting this in is a good habit – allows you to add features while keeping the user up to date
    - `man -h`
- `fopen/fclose`
  - Always error check!
- `malloc()`
  - Error check
  - Free everything you allocate
- Global variables
  - Namespace pollution
  - If you must, make them private: `static int foo;`
Questions and References Slide

- Rubber Duck 1
- Rubber Duck Debugging on Wiki
- Florentijn Hofman’s Duck
- Good sigsuspend() reference
- Indirection on Wiki
- Pictures stolen from lecture slides
- Stole from 15-410 Virtual Memory Slides
  - Lectures reside [here](#)
  - BTW, Prof. Eckhardt is super cool