Processes, Signals, I/O, Shell Lab and Midterm Review

15-213: Introduction to Computer Systems
Recitation 6: June 26, 2014
Agenda

- News
- **Processes**
  - Overview
  - Important functions
- Signals
  - Overview
  - Important functions
  - Race conditions
- **I/O Intro**
- **Shell Lab Tips**
Processes

- An instance of an executing program
- Abstraction provided by the operating system
- Properties
  - Have a process ID (pid) and process group ID (pgid)
  - Private state – memory, registers, etc.
  - Shared state - such as open file table
  - Become zombies when finished running (why?)
Process: fork()

- Prototype:
  
  ```c
  pid_t fork(void);
  ```

- Clones the current process. The new process gets a new pid, but the same pgid.

- The new process is an exact duplicate of the parent’s state. It has its own stack, own registers, etc.

- It has its own file descriptors (but the files themselves are shared).

- Called once, returns twice (once in the parent, once in the child).

- Return value in child is 0, child’s pid in parent. (This is how the parent can keep track of who its child is.)

- Returns -1 in case of failure.

- After the fork, we do not know which process will run first, the parent or the child.
Process: execve()

- Prototype:
  ```c
  int execve(const char *filename, char *const argv[],
              char *const envp[]);
  ```
- Replaces the current process with a new one. The binary corresponding to `filename` will be run by current process.
- Called once; does not return (or returns -1 on failure).
- `fork() + execve()` creates a new process and runs a new binary on it. This is the usual way of running a new process.
Process: exit()

- Prototype:
  
  ```c
  void exit(int status);
  ```

- Immediately terminates the process that called it. The process goes to Zombie state.

- `status` is normally the return value of `main()`.

- The OS frees the resources (heap, file descriptors, etc.) but not its exit status. It remains in the process table to await its reaping.

- Zombies are reaped when their parents read their exit status. (If the parent is dead, this is done by `init`.) Then its pid can be reused.
Process: waitpid()

- Prototype:
  
  ```c
  pid_t waitpid(pid_t pid, int *status, int options);
  ```

- The wait family of functions allows a parent to know when a child has changed state (e.g., terminated).

- `waitpid` returns when the process specified by `pid` terminates.

- `pid` must be a direct child of the invoking process.

- If `pid` = -1, it will wait for any child of the current process.

- Return value: the pid of the child it reaped.

- Writes to `status`: information about the child’s status.

- `options` variable: used to modify `waitpid`’s behavior.
  - `WNOHANG`: keep executing caller until a child terminates.
  - `WCONTINUED`: report continued children too
Processes – setpgid()

- Prototype:
  setpgid(pid_t pid, pid_t pgid)

  - Sets the process group id (pgid) of the given pid
  - If *pid*=0, setpgid is applied to the calling process
  - If *pgid*=0, setpgid uses pgid=pid of the calling process
  - Children inherit the pgid of their parents by default
process 5 can reap processes 8 and 213, but not 500. Only process 213 can reap process 500.
Concurrency!

```c
pid_t child_pid = fork();

if (child_pid == 0) {
    printf("Child!\n");
    exit(0);
}
else {
    printf("Parent!\n");
}

Output?
```
Concurrency!

pid_t child_pid = fork();

if (child_pid == 0) {
    printf("Child!\n");
    exit(0);
}

else {
    printf("Parent!\n");
}

Two possible Outcomes:

• Child!
  Parent!

• Parent!
  Child
Concurrent concurrency:

```c
pid_t child_pid = fork();
if (child_pid == 0) {
    printf("Child!\n");
    exit(0);
}
else {
    printf("Parent!\n");
}
```

**Two possible Outcomes:**

- Child!
  Parent!
- Parent!
  Child

```c
int status;
pid_t child_pid = fork();
if (child_pid == 0) {
    printf("Child!\n");
    exit(0);
}
else {
    waitpid(child_pid,&status, 0);
    printf("Parent!\n");
}
else {
    waitpid(child_pid,&status, 0);
    printf("Parent!\n");
}
```

**Only one possible Outcome:**

- Child!
  Parent!
Agenda

- News
- Processes
  - Overview
  - Important functions
- Signals
  - Overview
  - Important functions
  - Race conditions
- I/O Intro
- Shell Lab Tips
Signals

- Signals are the basic way processes communicate with each other. They notify a process that an event has occurred (for example, that its child has terminated).
- They are sent several ways: Ctrl-C, Ctrl-Z, kill()
- Signals are asynchronous. They aren’t necessary received immediately; they’re received right after a context switch.
- They are non-queuing.
  - There is only one bit in the context per signal
  - If 100 child processes die and send a SIGCHLD, the parent may still only receive one SIGCHLD
- Three possible ways to react when a signal is received:
  - Ignore the signal (do nothing)
  - Terminate the process (with optional core dump)
  - Catch the signal by executing a user-level function called signal handler
Sending a signal

- `kill(pid_t id, int sig)`
  - If id positive, sends signal sig to process with pid=id
  - If id negative, sends signal sig to all processes with pgid=-id
kill() with a positive PID will send the signal only to the process with that ID.

```
kill(8, SIGINT);
```
kill() with a negative PID will send the signal to all processes with that group ID.
Handling signals

- `signal(int signum, sighandler_t handler)`
  - Specifies a handler function to run when signum is received
  - `sighandler_t` means a function which takes in one int argument and is void (returns nothing)
  - When a signal is caught using the handler, its default behavior is ignored
  - The handler can interrupt the process at any time, even while either it or another signal handler is running
  - Control flow of the main program is restored once it's finished running
  - SIGKILL, SIGSTOP cannot be caught
Caveat

• Remember Signals are received asynchronously.
• Signal handlers can be called anytime when the program is running.
• Concurrency bug?
  • What if main() and signal_handler() access a common data?
  • A typical scenario in your shell lab
• Solution: Block Signals
Signals (contd..)

- **Blocking Signals**
  - Processes can choose to block signals using a signal mask
  - While a signal is blocked, the signal will be still delivered to the process but keep it pending
  - No action will be taken until the signal is unblocked
  - Implemented using `sigprocmask()`

- **Waiting for Signals**
  - Sometimes, a process needs to wait for a signal to be received.
  - Implemented using `sigsuspend()`
Blocking Signals – sigprocmask()

- `sigprocmask(int option, const sigset_t* set, sigset_t* oldSet)`
  - Updates the mask of blocked/unblocked signals using the handler signal set
  - Blocked signals are ignored until unblocked
    - Process only tracks whether it has received a blocked signal, not the count
    - Getting SIGCHILD 20 times while blocked then unblocking will only run its handler once
  - `option`: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
  - Signal mask's old value is written into oldSet
Waiting for Signals – sigsuspend()" s

- sigsuspend(sigset_t *tempMask)
  - Temporarily replaces the signal mask of the process with tempMask
  - Sigsuspend will return once it receives an unblocked signal (and after its handler has run)
  - Good to stop code execution until receiving a signal
  - Once sigsuspend returns, it automatically reverts the process signal mask to its old value
Signals – sigsetops

- A family of functions used to create and modify sets of signals. E.g.,
  - `int sigemptyset(sigset_t *set);`
  - `int sigfillset(sigset_t *set)`
  - `int sigaddset(sigset_t *set, int signum);`
  - `int sigdelset(sigset_t *set, int signum);`

- These sets can then be used in other functions.

- [http://linux.die.net/man/3/sigsetops](http://linux.die.net/man/3/sigsetops)

- Remember to pass in the address of the sets, not the sets themselves
Race Conditions

- Race conditions occur when sequence or timing of events are random or unknown
- Signal handlers will interrupt currently running code
- When forking, child or parent may run in different order
- If something can go wrong, it will
  - Must reason carefully about the possible sequence of events in concurrent programs
Race Conditions

// sigchld handler installed

pid_t child_pid = fork();

if (child_pid == 0){
    /* child comes here */
    execve(......);
}

else{
    add_job(child_pid);
}

void sigchld_handler(int signum)
{
    int status;
    pid_t child_pid =
    waitpid(-1, &status, WNOHANG);
    if (WIFEXITED(status))
        remove_job(child_pid);
}

• Does add_job() or remove_job() come first?
• Where can signals be blocked to ensure correctness?
Agenda

■ News
■ Processes
  ▪ Overview
  ▪ Important functions
■ Signals
  ▪ Overview
  ▪ Important functions
  ▪ Race conditions
■ I/O Intro
■ Shell Lab Tips
Unix I/O

- All Unix I/O, from network sockets to text files, are based on one interface.
- A file descriptor is what’s returned by `open()`.
  ```c
  int fd = open("/path/to/file", O_RDONLY);
  ```
- It’s just an int, but you can think of it as a pointer into the file descriptor table.
- Every process starts with three file descriptors by default:
  - 0: STDIN
  - 1: STDOUT
  - 2: STDERR.
- Every process gets its own file descriptor table, but processes share the open file table and v-node table.
Unix I/O – dup2() 

- Prototype: 
  
  ```c
  int dup2(int oldfd, int newfd);
  ```

- `newfd` becomes a copy of `oldfd`;
- Read/write on `newfd` will access the file corresponding to `oldfd`.
- This is handy for implementing I/O redirection in shelllab.
Unix I/O – Practice Problem

```c
int main()
{
    int fd = open("ab.txt", O_RDONLY);
    char c;
    fork();
    read(fd,&c,1); //Read one character from the file
    printf("%c\n",c); //Print the character
}
```

- Assume the file ab.txt contains “ab”
- What do the file tables look like?
- What's the output?
- What if the process forked before opening the file?
Agenda

- News
- Processes
  - Overview
  - Important functions
- Signals
  - Overview
  - Important functions
  - Race conditions
- I/O Intro
- Shell Lab Tips
Shell Lab Tips

- There's a lot of starter code
  - Look over it so you don't needlessly repeat work

- Use the reference shell to figure out the shell's behavior
  - For instance, the format of the output when a job is stopped

- Use sigsuspend, not waitpid, to wait for foreground jobs
  - You will lose points for using tight loops (while(1) {}), sleeps to wait for the foreground
Shell Lab Tips

- Shell requires SIGINT and SIGSTP to be forwarded to the foreground job (and all its descendants) of the shell
  - How could process groups be useful?
- dup2 is a handy function for the last section, I/O redirection
- SIGCHILD handler may have to reap multiple children per call
- Try actually using your shell and seeing if/where it fails
  - Can be easier than looking at the driver output
Questions?
Midterm Review

15-213: Introduction to Computer Systems
Recitation 6b: June 26, 2014
Marjorie Carlson
Agenda

- Midterm Logistics

- A **Non-Comprehensive** Review of Midterm Topics:
  - Some skills you should have
  - Sample exam problems
Midterm Logistics

■ Pittsburgh students:
  - Monday & Wednesday 10 a.m. – 10 p.m.
  - Tuesday 10 a.m. – 3 p.m.
  - Gates-Hillman 3000

■ Distance students:
  - Any place & time between Mon. 10 am and Wed. 10 pm (EDT).
    We’ll email you a URL and password just beforehand.

■ Everybody:
  - You’ll have unlimited time, though it’s intended to take 80 mins.
  - You will be able to print questions, if you want.
Midterm Logistics

■ What to bring?
  ▪ Your exam password, which will be emailed to you this weekend.
  ▪ A cheat sheet – ONE double-sided 8 ½ x 11 paper; cannot include pre-worked problems. And blank scratch paper, if you wish.
  ▪ No other resources are to be used during the exam (calculator, compiler, phone, textbook, Google...)

■ What to study?
  ▪ Chapters 1-3 and Chapter 6
  ▪ Lectures through cache

■ How to study?
  ▪ Read each chapter (preferably multiple times) and do the problems in the book.
  ▪ Do problems from previous exams (including cache from midterm 2, if you go back that far).
Agenda

- Midterm Logistics

- A Non-Comprehensive Review of Midterm Topics:
  - Bits, bytes & ints
  - Floats
  - Structs (sizes & alignment of data)
  - Assembly loops
  - The stack
  - Array access
  - Cache
Bits, Bytes & Integers

Some suggested skills:

- Know how to do basic bitshifting by hand.

- If you represent numbers in $w$ bits:
  - what’s UMax (the largest unsigned number)? $2^w - 1$
  - what’s Tmax (the largest signed number)? $2^{w-1} - 1$
  - what’s Tmin (the most negative signed number)? $-2^{w-1}$

- Know how to calculate $-x$ in two’s complement. $\sim x + 1$

- Know what happens when signed are cast to unsigned and vice versa, and when it occurs implicitly.
Floating Point

- **Some suggested skills:**
  - Know how to represent a fraction in binary scientific notation.
  
  
  - Be able to convert between this format:

    \[
    (-1)^s \times M \times 2^E
    \]

    and this format:

    
    \[
    \begin{array}{c|c|c}
    s & \text{exp} & \text{frac} \\
    \hline
    \end{array}
    \]

    - Be able to compute bias: \(2^{k-1} - 1\) where \(k\) is the number of exp bits.

    - Know how to convert from one mini float standard to another — whether to convert normalized to denormalized & vice versa, when to round up/down/to even, and when numbers round to \(0/\infty\).
## Floating Point

<table>
<thead>
<tr>
<th>Represents:</th>
<th>Normalized</th>
<th>Denormalized</th>
<th>Special Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most numbers</td>
<td>Tiny numbers</td>
<td>Infinity, NaN</td>
<td></td>
</tr>
<tr>
<td>Exponent bits:</td>
<td>Not those →</td>
<td>000...000</td>
<td>111...111</td>
</tr>
<tr>
<td>E =</td>
<td>exp – bias</td>
<td>1 – bias</td>
<td>Represents +/- ∞ if frac is all 0s, otherwise NaN</td>
</tr>
<tr>
<td>M =</td>
<td>1.frac</td>
<td>0.frac</td>
<td></td>
</tr>
</tbody>
</table>

- **Floating Point Rounding:**
  - Round **up** if the spilled bits are greater than half
  - Round **down** if the spilled bits are less than half
  - Round **to even** if the spilled bits are exactly equal to half
Structs: Sizes & Alignment

■ Some suggested skills:

- For each major data type (e.g., char, short, int, pointer, long, float, double), know its **size** and its **alignment rules** in both **32-bit** and **64-bit systems**.

- Understand where padding will occur, both within a struct and at the end of a struct.

- Be able to show how to pack a struct more efficiently.
Assembly Loops

- Some suggested skills:
  - Know the basic assembly instructions, jump codes, and suffixes indicating size.
  - Get comfortable with the order of operands!
    - sub %edx,%eax \implies %eax = %eax-%edx
  - Get comfortable with dereferencing and addresses:
    - %edx vs. (%edx)
    - D(Rb, Ri, S) \implies Mem[Reg[Rb] + S * Reg[Ri] + D]
  - Hint: if statements often end up “backwards” in Assembly
    
    ```assembly
    if (n < 5)        cmp $0x5,%edx
        do something;  jge <something else>
    else
        do something else;
    [do something]
    ```
The Stack

- **Some suggested skills:**
  - Know how arguments are passed to a function.
    - 32-bit?
    - 64-bit?
  - Know where to find the return value from a function.
    - 32-bit?
    - 64-bit?
  - Know how these instructions modify the stack.
    - call
    - leave
    - ret
    - pop
    - push
Array Access

A suggested method:

- Understand how assembly turns a 2D array into a 1D array by using the width of the array as a multiplier.

| [0][0] = [0] | [0][1] = [1] | [0][2] = [2] | [0][3] = [3] |
| [1][0] = [4] | [1][1] = [5] | [1][2] = [6] | [1][3] = [7] |
| [2][0] = [8] | [2][1] = [9] | [2][2] = [10] | [2][3] = [11] |

\[
[0][2] = (0 \times 4) + 2 = 2 \\
[1][3] = (1 \times 4) + 3 = 7 \\
[2][1] = (2 \times 4) + 1 = 9 \\
[i][j] = i \times \text{width of array} + j
\]

- So, if you have the assembly for a 2D array access, you can figure out the width of an array.
Cache

- **Dimensions: S, E, B**
  - S: Number of sets
  - E: Associativity – number of lines per set
  - B: Block size – number of bytes per block (1 block per line)

- **Some suggested skills:**
  - Given values for S/s, E, B/b, and m:
    - Figure out which addresses map to which set
    - Will a given access result in a hit? a miss? an eviction?
  - Be able to estimate the hit rate or miss rate for a given piece of C code.
For this problem, assume the following:

- We are running code on an 8-bit machine using two’s complement arithmetic for signed integers.
- `short` integers are encoded using 4 bits.
- Sign extension is performed whenever a `short` is cast to an `int`.

The following definitions are used in the table below:

```c
short sa = -6;
int b = 2*sa;
short sc = (short)b;
int x = -64;
unsigned ux = x;
```

Fill in the empty boxes in the table. If the expression is cast to or stored in a `short`, use a 4-bit binary representation. Otherwise assume an 8-bit binary representation. The first 2 lines are given to you as examples, and you need not fill in entries marked with “—”.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Decimal Representation</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0</td>
<td>0000 0000</td>
</tr>
<tr>
<td>(short)0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>−</td>
<td>−17</td>
<td></td>
</tr>
<tr>
<td>−</td>
<td>0010 1001</td>
<td></td>
</tr>
<tr>
<td>sa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tmax − Tmin</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Floating point encoding.** In this problem, you will work with floating point numbers based on the IEEE floating point format. We consider two different 6-bit formats:

**Format A:**

- There is one sign bit $s$.
- There are $k = 3$ exponent bits. The bias is $2^{k-1} - 1 = 3$.
- There are $n = 2$ fraction bits.

**Format B:**

- There is one sign bit $s$.
- There are $k = 2$ exponent bits. The bias is $2^{k-1} - 1 = 1$.
- There are $n = 3$ fraction bits.

For formats A and B, please write down the binary representation for the following (use round-to-even). Recall that for denormalized numbers, $E = 1 - \text{bias}$. For normalized numbers, $E = e - \text{bias}$.

<table>
<thead>
<tr>
<th>Value</th>
<th>Format A Bits</th>
<th>Format B Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>0 000 00</td>
<td>0 00 000</td>
</tr>
<tr>
<td>One</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1/2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$11/8$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
struct {
    char a[9];
    short b[3];
    float c;
    char d;
    int e;
    char *f;
    short g;
} foo;

A. Show how the struct above would appear on a 64-bit ("x86_64") Windows machine (primitives of size $k$ are $k$-byte aligned). Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x’s to indicate bytes that are allocated in the struct but are not used.

```
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |                   |
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
```

B. Rearrange the above fields in `foo` to conserve the most space in the memory below. Label the bytes that belong to the various fields with their names and clearly mark the end of the struct. Use hatch marks or x’s to indicate bytes that are allocated in the struct but are not used.
Assembly Loops

```c
void mystery(int *array, int n) {
    int i;
    for (________; _______; ______) {
        if (__________________)
            ________;
    }
}
```
Given assembly code of foo() and bar(), draw a detailed picture of the stack, starting with the caller invoking foo(3, 4, 5).

Value of %ebp when foo is called: 0xffffd858
Return address in function that called foo: 0x080483c9

Stack addresss

<table>
<thead>
<tr>
<th>Address</th>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffd850</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>0xffffd84c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffd848</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffd844</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffd840</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffd83c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffd838</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffd834</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffffd830</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From fall 2012; answers here.
int bar (int a, int b) {
    return a + b;
}

int foo(int n, int m, int c) {
    c += bar(m, n);
    return c;
}

08048374 <bar>:
8048374:  55           push %ebp
8048375:  89 e5       mov %esp,%ebp
8048377:  8b 45 0c  mov 0xc(%ebp),%eax
804837a:  03 45 08  add 0x8(%ebp),%eax
804837d:  5d           pop %ebp
804837e:  c3           ret

0804837f <foo>:
804837f:  55           push %ebp
8048380:  89 e5       mov %esp,%ebp
8048382:  83 ec 08  sub $0x8,%esp
8048385:  8b 45 08  mov 0x8(%ebp),%eax
8048388:  89 44 24 04  mov %eax,0x4(%esp)
804838c:  8b 45 0c  mov 0xc(%ebp),%eax
804838f:  89 04 24  mov %eax,(%esp)
8048392:  e8 dd ff ff ff  call 8048374 <bar>
8048397:  03 45 10  add 0x10(%ebp),%eax
804839a:  c9           leave
804839b:  c3           ret
int array1[H][J];
int array2[J][H];

int copy_array(int x, int y) {
array2[y][x] = array1[x][y];

    return 1;
}

Suppose the above C code generates the following x86-64 assembly code:

# On entry:
#  %edi = x
#  %esi = y
#

 copy_array:
    movslq %esi,%rsi
    movslq %edi,%rdi
    movq  %rsi, %rax
    salq  $4, %rax
    subq  %rsi, %rax
    addq  %rdi, %rax
    leaq  (%rdi,%rdi,2), %rdi
    addq  %rsi, %rdi
    movl  array1(,%rdi,4), %edx
    movl  %edx, array2(,%rax,4)
    movl  $1, %eax
    ret
The Hit or Miss Question

Given a 32-bit Linux system that has a 2-way associative cache of size 128 bytes with 32 bytes per block. Long longs are 8 bytes. For all parts, assume that table starts at address 0x0.

```c
int i;
int j;
long long table[4][8];
for (j = 0; j < 8; j++) {
    for (i = 0; i < 4; i++) {
        table[i][j] = i + j;
    }
}
```

B. This problem refers to code sample above. In the table below write down in each space whether that element’s access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What is the miss rate of this code sample?

From spring 2011; answers here.