18600 Final Exam Review Recitation

General Topics

- The Big Picture
- Assembly
- Pipeline/Superscalar/Memory Hierarchy & Program Optimization
- Exceptional Control Flow
- Virtual Memory
- Dynamic Memory Allocation
- Parallel Architecture/Cache Coherence
- System Level I/O, Network Programming
- Concurrent Programming

Note: other topics may appear on the final exam!

Logistics for Final Exam

Thursday, December 15

5:30pm - 8:30pm (ET)

- Closed book, paper exam

Note Sheet Allowed - ONE double sided 8 ½ x 11 paper

- No worked out problems on that sheet

The Big Picture

Multiple choice questions focused around concepts covered in lab assignments, and recitations.

The Big Picture - Sample Question 1

Dynamic memory is used because:

1. The heap is significantly faster than the stack.
2. The stack is prone to corruption from buffer overflows.
3. Storing data on the stack requires knowing the size of that data at compile time
4. None of the above.

Answer: 3

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
  - For data structures whose size is only known at runtime.

The Big Picture - Sample Question 2

Which of the following socket API is not used by the server:

1. bind
2. connect
3. getaddrinfo
4. accept

Answer: 2

int connect(int socket, struct sockaddr *address, socklen_t address_len);

- attempt to connect to the specified IP address and port described in address
- used by clients

Assembly

- **Similar problems to those seen in the midterm**
  - Reading assembly, answering questions about the layout of the stack
- **General Advice**
  - Brush up on Assembly syntax
  - Understand x86-64 stack conventions
  - Be able to draw a stack diagram
  - Study Attack Lab
  - Study the Midterm question

Assembly: Practice Problem

```
void fill(char *dest, char *src, int a)
{
    if (a != 0xdeadbeef)
    {
        fill (dest,src,0xdeadbeef);
        return;
    }
    strcpy(dest,src);
}

void getbuf(void)
{
    int buf[2]; //way too small
    fill((char*)(&buf[0]),"complexes",0x15213);
    printf("0x%.8x \n",buf[0]);
    printf("0x%.8x \n", buf[1]);
    printf("0x%.8x \n",buf[6]);
}

int main(void)
{
    getbuf();
    return 0;
}
```

```
Dump of assembler code for function getbuf:
   0x00000000004005e2 <+0>:     sub    $0x18,%rsp
   0x00000000004005e6 <+4>:     mov    %rsp,%rax
   0x00000000004005e9 <+7>:     mov    $0x15213,%edx
   0x00000000004005ee <+12>:    mov    $0x40074c,%esi
   0x00000000004005f3 <+17>:    mov    %rax,%rdi
   0x00000000004005f6 <+20>:    callq  0x400590 <fill>
   0x00000000004005fb <+25>:    mov    (%rsp),%eax
   0x00000000004005fe <+28>:    mov    %eax,%esi
   0x0000000000400600 <+30>:    mov    $0x400756,%edi
   0x0000000000400605 <+35>:    mov    $0x0,%eax
   0x000000000040060a <+40>:    callq  0x4003f8 <printf@plt>
   0x000000000040060f <+45>:    mov    0x4(%rsp),%eax
   0x0000000000400613 <+49>:    mov    %eax,%esi
   0x0000000000400615 <+51>:    mov    $0x400756,%edi
   0x000000000040061a <+56>:    mov    $0x0,%eax
   0x000000000040061f <+61>:    callq  0x4003f8 <printf@plt>
   0x0000000000400624 <+66>:    mov    0x18(%rsp),%eax
   0x0000000000400628 <+70>:    mov    %eax,%esi
   0x000000000040062a <+72>:    mov    $0x400756,%edi
   0x000000000040062f <+77>:    mov    $0x0,%eax
   0x0000000000400634 <+82>:    callq  0x4003f8 <printf@plt>
   0x0000000000400639 <+87>:    add    $0x18,%rsp
   0x000000000040063d <+91>:    retq

Dump of assembler code for function main:
   0x000000000040063e <+0>:     sub    $0x8,%rsp
   0x0000000000400642 <+4>:     mov    $0x0,%eax
   0x0000000000400647 <+9>:     callq  0x4005e2 <getbuf>
   0x000000000040064c <+14>:    mov    $0x0,%eax
   0x0000000000400651 <+19>:    add    $0x8,%rsp
   0x0000000000400655 <+23>:    retq
```

Assembly: Practice Problem

Assume that immediately before the call to `getbuf` in `main`, the register `%rsp` contains `0x7fffd178`.

You might need the following ascii values (in hex) for the different characters --------->

Remember to keep in mind that addresses are 64 bits long

c	0x63
e	0x65
l	0x6c
m	0x6d
o	0x6f
p	0x70
s	0x73
x	0x78

- What is the address of the buffer `buf` in `getbuf`?

  Ans: 0x7fffd158 (remember, 8 bytes used for return address)

- Immediately after `fill` returns, what are the values that are printed by each of the three print statements? (The format string `"0x%.8x"` prints in a hexadecimal format with 8 digits, zero-padded.)

  Print statement 1 : 0x706d6f63
  Print statement 2 : 0x6578656c
  Print statement 3 : 0x0040064c

- Is the stack frame corrupted?

  Ans: No

- If your answer above is Yes, enter 0 below. Otherwise, enter the minimum number of additional characters which must be written to corrupt the stack.

  Ans: 15

Stack Diagram

0x7fffd178	<end of main frame>
0x7fffd170	Ret addr: 0x0040064c
0x7fffd168	
0x7fffd160	0x0073
0x7fffd158	buf = 0x6578656c706d6f63

Program Optimization

- Memory Aliasing
- General Optimizations

Program Optimization - Memory Aliasing

Case where two pointers may designate the same memory location is referred to as **memory aliasing.**

Let's take the simple example of **strcpy(char *dest, char *src)**

```
char *strcpy(char *dest, const char *src)
{
  unsigned i;
  for (i=0; src[i] != '\0'; ++i)
    dest[i] = src[i];
  dest[i] = '\0';
  return dest;
}
```

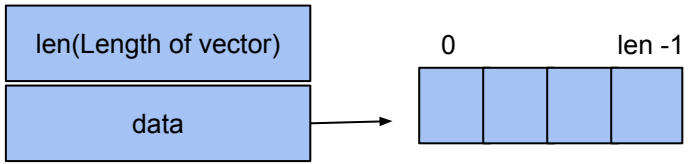
If **dest = src + 1**: the result would be different from a character-by-character copy.

Compiler can't assume that **src** and **dest** do not overlap; generates more assembly to take care of cases where there is an overlap.

Aside: In actual implementation: **restrict** keyword is used to tell the compiler there is no overlap; leads to more efficient code.

Program Optimization - General Optimizations

- CPE: Cycles Per Element is a useful metric to measure program performance
- We illustrate a series of optimization technique for calculating the sum of a vector
- In all the examples get_vec_element() does bound checking on the vector
- A vector is represented by header information plus an array of designated length



Initial Implementation

```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest + val;
    }
}
```

What is the simplest optimization on this program ?

Improve efficiency of loop test

```
void combine2(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest + val;
    }
}
```

Move function calls that do not change the return value out of the loop.

What else can be optimized ? Do we really need the bound check ?

Eliminate all function calls within the loop if possible

```
void combine3(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        *dest = *dest + data[i];
    }
}
```

Can we avoid some memory references ?

Accumulate result in temporary

```
void combine4(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = 0;
    for (i = 0; i < length; i++) {
        acc = acc + data[i];
    }
    *dest = acc;
}
```

Holding accumulated value in local variable avoids repeated access to memory.

We now have a loop. Can you think of a familiar optimization ? ;)

Loop Unrolling

```
/* Unroll loop by 2 */
void combine5(vec_ptr v, data_t *dest)
{
     long int i;
     long int length = vec_length(v);
     long int limit = length-1;
     data_t *data = get_vec_start(v);
     data_t acc = 0;
     /* Combine 2 elements at a time */
     for (i = 0; i < limit; i+=2) {
          acc = (acc + data[i]) + data[i+1];
     }
     /* Finish any remaining elements */
     for (; i < length; i++) {
          acc = acc + data[i];
     }
     *dest = acc;
}
```

Anything more ?

Can you think of strategies that make use of parallelism in superscalar processors ?

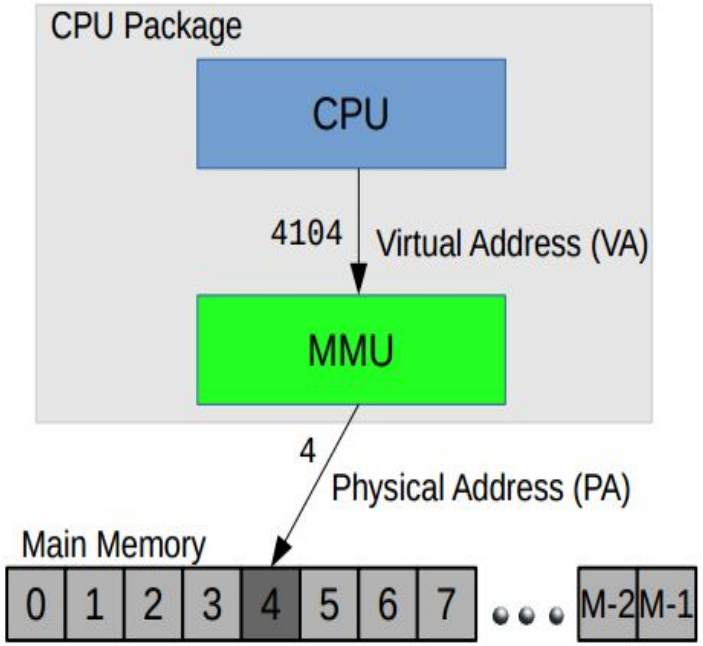
Loop Unrolling 2

```
 void combine6(vec_ptr v, data_t *dest)
{
        long int i, length = vec_length(v);
        long int limit = length-1;
        data_t *data = get_vec_start(v);
        data_t acc0 = 0;
        data_t acc1 = 0;
        /* Combine 2 elements at a time */
        for (i = 0; i < limit; i+=2) {
                acc0 = acc0 + data[i];
                acc1 = acc1 + data[i+1];
        }
        for (; i < length; i++) {
                acc0 = acc0 + data[i];
        }
        *dest = acc0 + acc1;
}
```

Maintain multiple accumulators to make better use of multiple functional units

Virtual Memory

- Virtual memory is a memory management technique that maps memory addresses used by a program, called virtual addresses into physical addresses.
- Allows multiple programs to run in the same address range .
- Virtual addresses used by programs get mapped to the actual physical address in main memory by the memory-management unit.
- VM can be thought of as an array of N contiguous bytes on disk, with the M bytes of physical memory as cache.



Address Translation

Virtual Address Space V = {0, 1, ..., N-1}

- virtual addresses are n bits long ($2^n = N$)

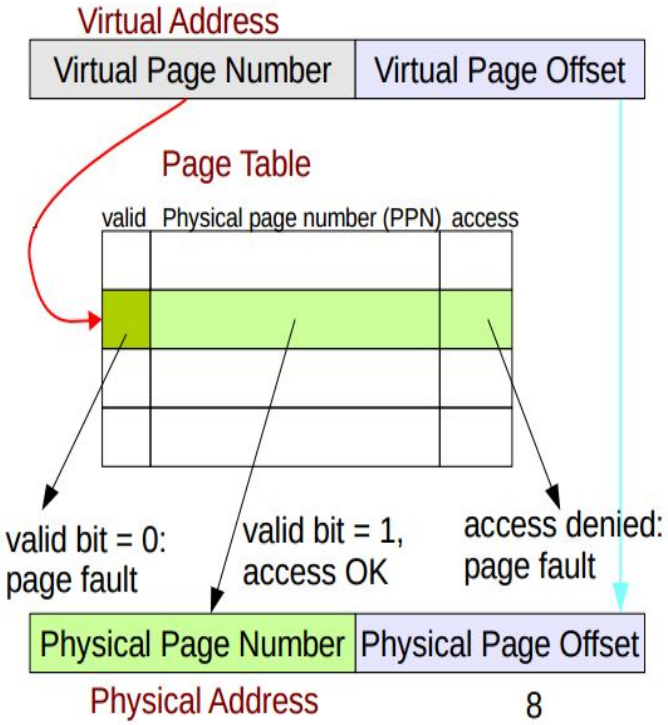
Physical Address Space P = {0, 1, ..., M-1}

- physical addresses are m bits long ($2^m = M$)

Memory is divided into "pages"

- page size is P bytes; the offset into a page is p bits ($2^p = P$)
- virtual page offset (VPO) and physical page offset (PPO) are the same!
  - no need to translate those bits

Page table is an array of entries which specify for each virtual page whether the page is in memory the physical page number, etc.



Address Translation Concepts

On a page fault

- CPU suspends the instruction that caused the fault
- Page is loaded in memory and marked as present in the page table entry

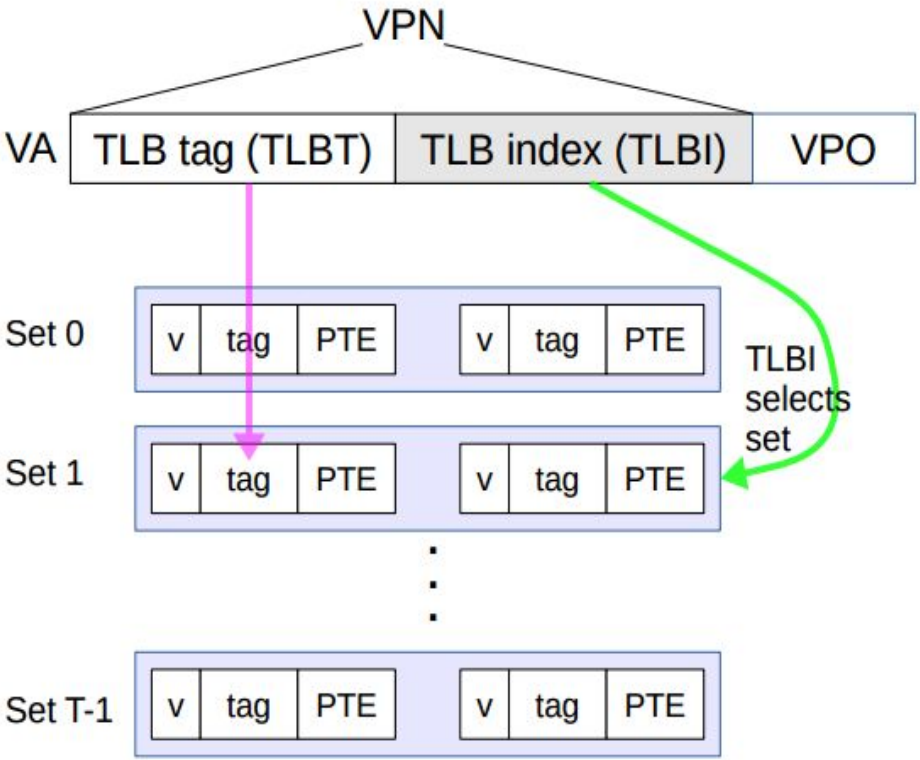
Multi-Level Page Tables

- A table of page tables
- Top-level page table (page directory) stays in memory
- Second level pages can be demand-paged like other data

Translation Lookaside Buffer (TLB)

- Hardware cache in MMU
- TLB hit eliminates memory access to get the page table entry

Translation Lookaside Buffer (TLB)



Q5

1 MB of virtual memory

256 KB of physical memory

4 KB page size

TLB: 8 entries, 2-way set associative

- How many bits are needed to represent the virtual address space?
- How many bits are needed to represent the physical address space?

- How many bits are needed to represent the page offset?
- How many bits are needed to represent the VPN?

- How many bits are in the TLB index?
- How many bits are in the TLB tag?

1 MB of virtual memory

256 KB of physical memory

4 KB page size

TLB: 8 entries, 2-way set associative

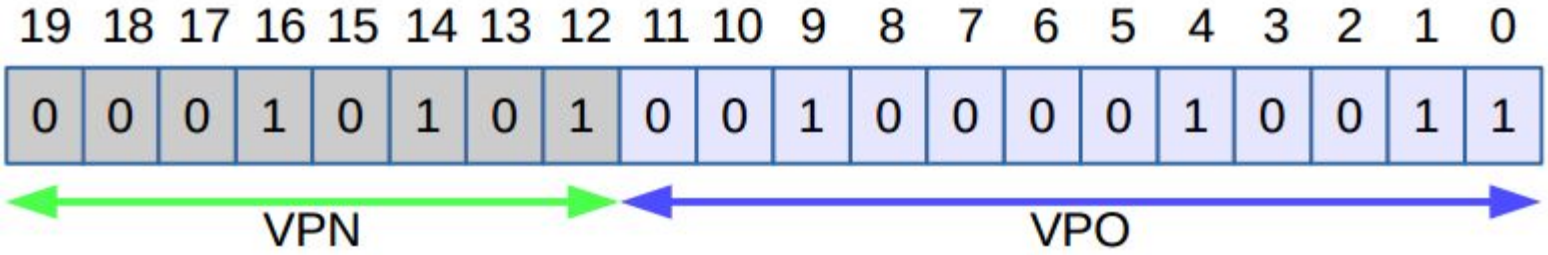
- How many bits are needed to represent the virtual address space?
- How many bits are needed to represent the physical address space?
  - **20 virtual (1MB = 2^{20}), 18 physical (256 KB = 2^{18})**
- How many bits are needed to represent the page offset?
- How many bits are needed to represent the VPN?
  - **12 offset bits (4 KB = 2^{12}), 8 bits for VPN (20-12)**
- How many bits are in the TLB index?
- How many bits are in the TLB tag?
  - **2 index bits (4 sets = 2^2), 6 tag bits (8-2)**

- Translate 0x15213, given the contents of the TLB and the first 32 entries of the page table below.

2-way set associative

Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

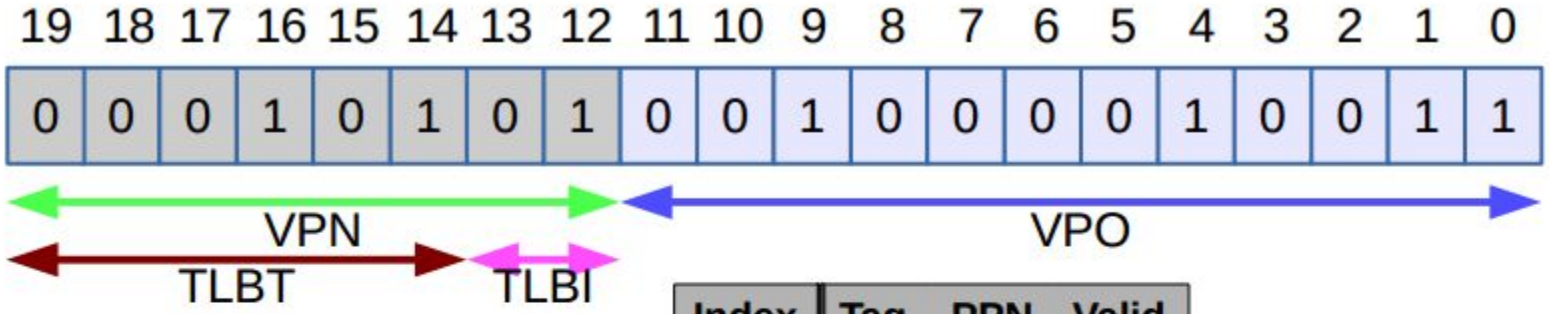
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	18	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1



VPN = ?

TLBI = ?

TLBT = ?



VPN = 0x15

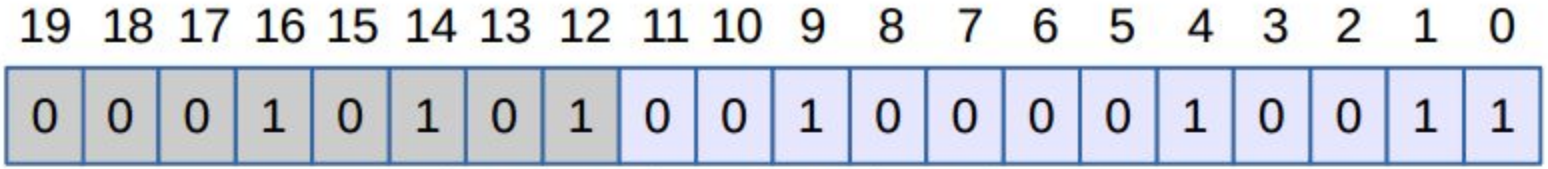
TLBI = 1

TLBT = 0x05

Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	**10**	**0F**	**1**
	0F	**1E**	**0**
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

TLB **Miss!**

We'll have to look it up in the page table....



VPN = 0x15

TLBI = 1

TLBT = 0x05

VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	18	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Page Table Hit
PPN = ?
Offset = ?

Physical Address:
?

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	0	0	1	0	0	0	0	1	0	0	1	1

VPN = 0x15

TLBI = 1

TLBT = 0x05

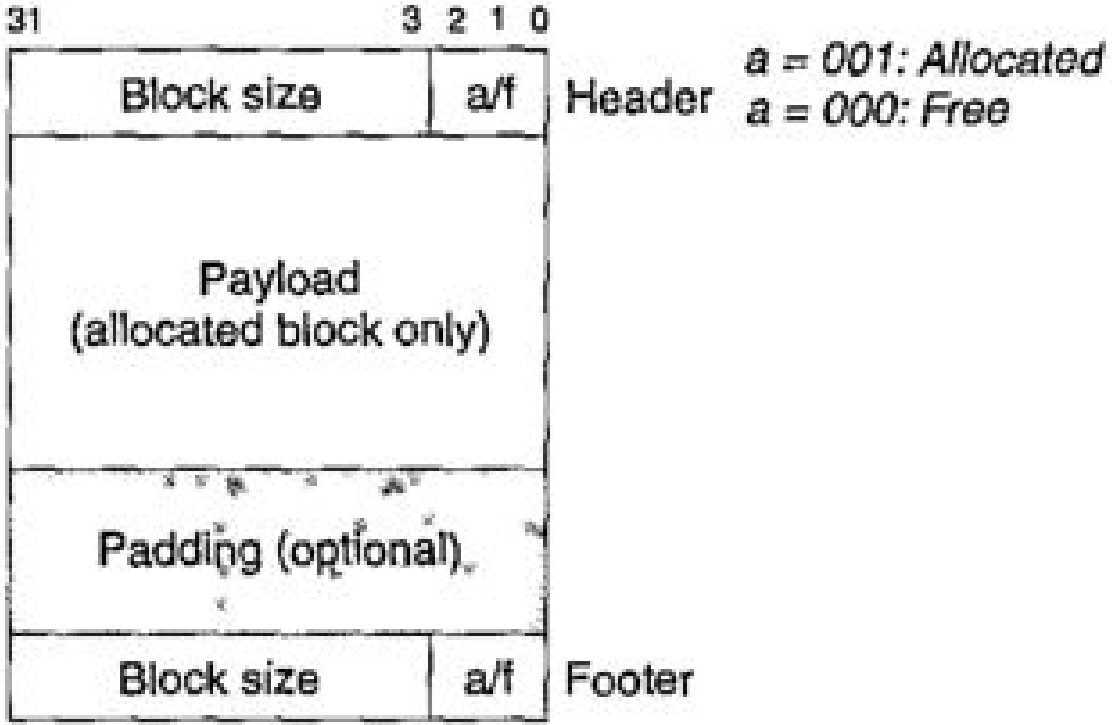
VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	**15**	**18**	**1**
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Page Table Hit
PPN = 0x18
Offset = 0x213

Physical Address:
0x18213

Question 7 Malloc

Typical format of heap block that uses a boundary tag



First fit vs best fit

First Fit fits data into memory by scanning from the beginning of available memory to the end,until the first free space which is at least big enough to accept the data is found. This space is then allocated to the data.

Best Fit tries to determine the best place to put the new data. The definition of 'best' may differ between implementations, but one example might be to try and minimise the wasted space at the end of the block being allocated - i.e.use the smallest space which is big enough.

What the heap looks like?

allocated block free block

48a	128f

- A single explicit free list
- All memory blocks have a size that is a multiple of 16 bytes and is at least 32 bytes
- All headers, footers and pointers are 8 bytes in size
- immediately coalesced after freeing
- Free blocks consist of a header
- All searches for free blocks start at the head of the list

First fit

ptr1 = malloc(48)

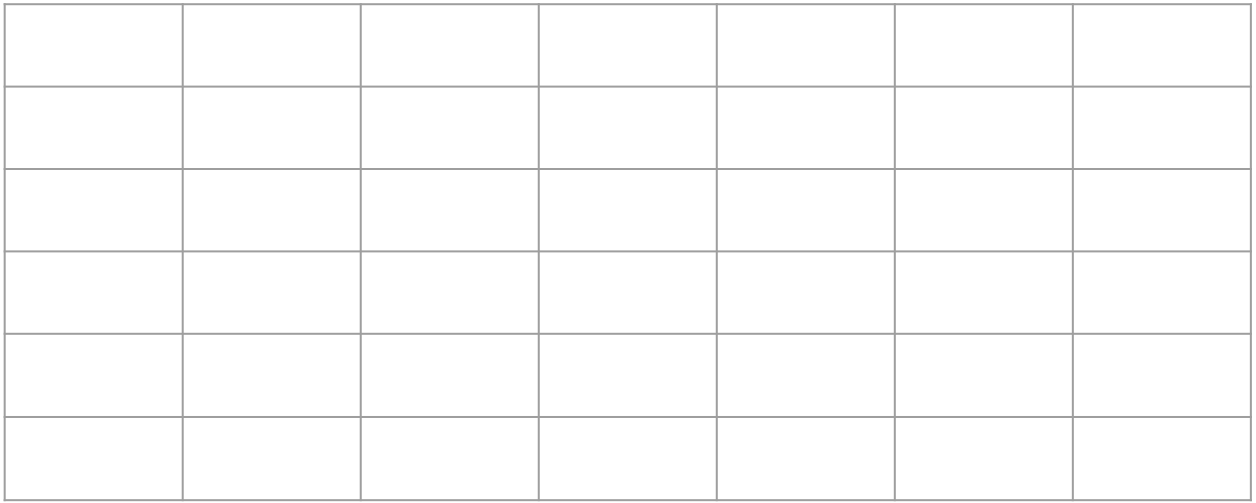
ptr2 = malloc(32)

ptr3 = malloc(32)

free(ptr3)

free(ptr1)

ptr4 = malloc(32)



What the heap looks like?

allocated block free block

48a	128f

- A single explicit free list
- All memory blocks have a size that is a multiple of 16 bytes and is at least 32 bytes
- All headers, footers and pointers are 8 bytes in size
- immediately coalesced after freeing
- Free blocks consist of a header
- All searches for free blocks start at the head of the list

First fit

ptr1 = malloc(48)

ptr2 = malloc(32)

ptr3 = malloc(32)

free(ptr3)

free(ptr1)

ptr4 = malloc(16)

64a						
64a	48a					
64a	48a	48a				
64a	48a	48f				
64f	48a	48f				
32a	32f	48a	48f			

What the heap looks like?

allocated block free block

48a	128f

- A single explicit free list
- All memory blocks have a size that is a multiple of 16 bytes and is at least 32 bytes
- All headers, footers and pointers are 8 bytes in size
- immediately coalesced after freeing
- Free blocks consist of a header
- All searches for free blocks start at the head of the list

ptr1 = malloc(48)

ptr2 = malloc(32)

ptr3 = malloc(32)

free(ptr3)

free(ptr1)

ptr4 = malloc(16)

……..

64a						
64a	48a					
64a	48a	48a				
64a	48a	48f				
64f	48a	48f				
32a	32f	48a	48f			

How about best fit?

Cache Coherence

Review lecture slides on:

1. MSI Cache Coherence Protocol
2. MESI Protocol

MESI Protocol

- Variation used in many Intel processors
- 4-State Protocol
  - **M**odified: <1,0,0…0>
  - **E**xclusive: <1,0,0,…,1>
  - **S**hared: <1,X,X,…,1>
  - **I**nvalid: <0,X,X,…X>
- Bus/Processor Actions
  - Same as MSI
- Adds *shared* signal to indicate if other caches have a copy

Processes, Signals and Threads (Practice Q)

Problem 8. (10 points):

Exceptional control flow: Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
int main()
{
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```

Processes, Signals and Threads

- **Recall that…**
  - fork() returns 0 for the child process
  - fork() returns the PID of the child for the parent process
  - wait(null) waits for a child process to finish
  - No guarantee of ordering of execution
    - Identify what orderings of executions can happen, and which cannot

Processes, Signals and Threads (Practice Q)

For each of the following strings, circle whether (Y) or not (N) this string is a possible output of the program. You will be graded on each sub-problem as follows:

- If you circle no answer, you get 0 points.
- If you circle the right answer, you get 2 points.
- If you circle the wrong answer, you get −1 points (so don't just guess wildly).

A. 01432	Y	N	**No (4 before 3 is impossible)**
B. 01342	Y	N	**Yes**
C. 03142	Y	N	**Yes**
D. 01234	Y	N	**No (2 must be last)**
E. 03412	Y	N	**Yes**

Concurrency

- Semaphores
- Mutex
- Race
- Deadlock
- Starvation
- Producer/Consumer

Problem 11. (9 points):

Synchronization. This problem is about using semaphores to synchronize access to a shared bounded FIFO queue in a producer/consumer system with an arbitrary number of producers and consumers.

- The queue is initially empty and has a capacity of 10 data items.
- Producer threads call the `insert` function to insert an item onto the rear of the queue.
- Consumer threads call the `remove` function to remove an item from the front of the queue.
- The system uses three semaphores: `mutex`, `items`, and `slots`.

Your task is to use P and V semaphore operations to correctly synchronize access to the queue.

A. What is the initial value of each semaphore?

`mutex` = _____

`items` = _____

`slots` = _____

B. Add the appropriate P and V operations to the psuedo-code for the `insert` and `remove` functions:

```
void insert(int item)
{
    /* Insert sem ops here */

    add_item(item);
    /* Insert sem ops here */

}
```

```
int remove()
{
    /* Insert sem ops here */

    item = remove_item();
    /* Insert sem ops here */

    return item;
}
```

Concurrency

Problem 11. (9 points):

Synchronization. This problem is about using semaphores to synchronize access to a shared bounded FIFO queue in a producer/consumer system with an arbitrary number of producers and consumers.

- The queue is initially empty and has a capacity of 10 data items.
- Producer threads call the `insert` function to insert an item onto the rear of the queue.
- Consumer threads call the `remove` function to remove an item from the front of the queue.
- The system uses three semaphores: `mutex`, `items`, and `slots`.

Your task is to use P and V semaphore operations to correctly synchronize access to the queue.

A. What is the initial value of each semaphore?

```
mutex = 1
items = 0
slots = 10
```

B. Add the appropriate P and V operations to the psuedo-code for the `insert` and `remove` functions:

```
void insert(int item)
{
    /* Insert sem ops here */
    P(slots);
    P(mutex);
    add_item(item);
    /* Insert sem ops here */
    V(mutex);
    V(items);
}
```

```
int remove()
{
    /* Insert sem ops here */
    P(items);
    P(mutex);
    item = remove_item();
    /* Insert sem ops here */
    V(mutex);
    V(slots);
    return item;
}
```

System Level I/O

File name	File contents
file_1.txt	file

The file file_1.txt contains the single word “file” with no white spaces
Assume that

- When each program finishes execution, the file contents will be reset to that shown above.
- All system calls will succeed and the files are in the same directory as the two programs.

```
/* buf is initialized to be all zeroes */
char buf[20] = {0};

int main(int argc, char* argv[]) {
    int fd1, fd2 = open("file_1.txt", O_RDONLY);

    fd1 = dup(fd2);
    read(fd2, buf, 3);
    close(fd2);
    read(fd1, &buf[3], 1);

    printf("%s", buf);

    /* Don't worry about file descriptors not being closed */
    return 0;
}
```

output to stdout from Program 1:	

System Level I/O

```
/* buf is initialized to be all zeroes */
char buf[20] = {0};

int main(int argc, char* argv[]) {
    int fd1, fd2 = open("file_1.txt", O_RDONLY);

    fd1 = dup(fd2);
    read(fd2, buf, 3);
    close(fd2);
    read(fd1, &buf[3], 1);

    printf("%s", buf);

    /* Don't worry about file descriptors not being closed */
    return 0;
}
```

output to stdout from Program 1:	file

Questions?