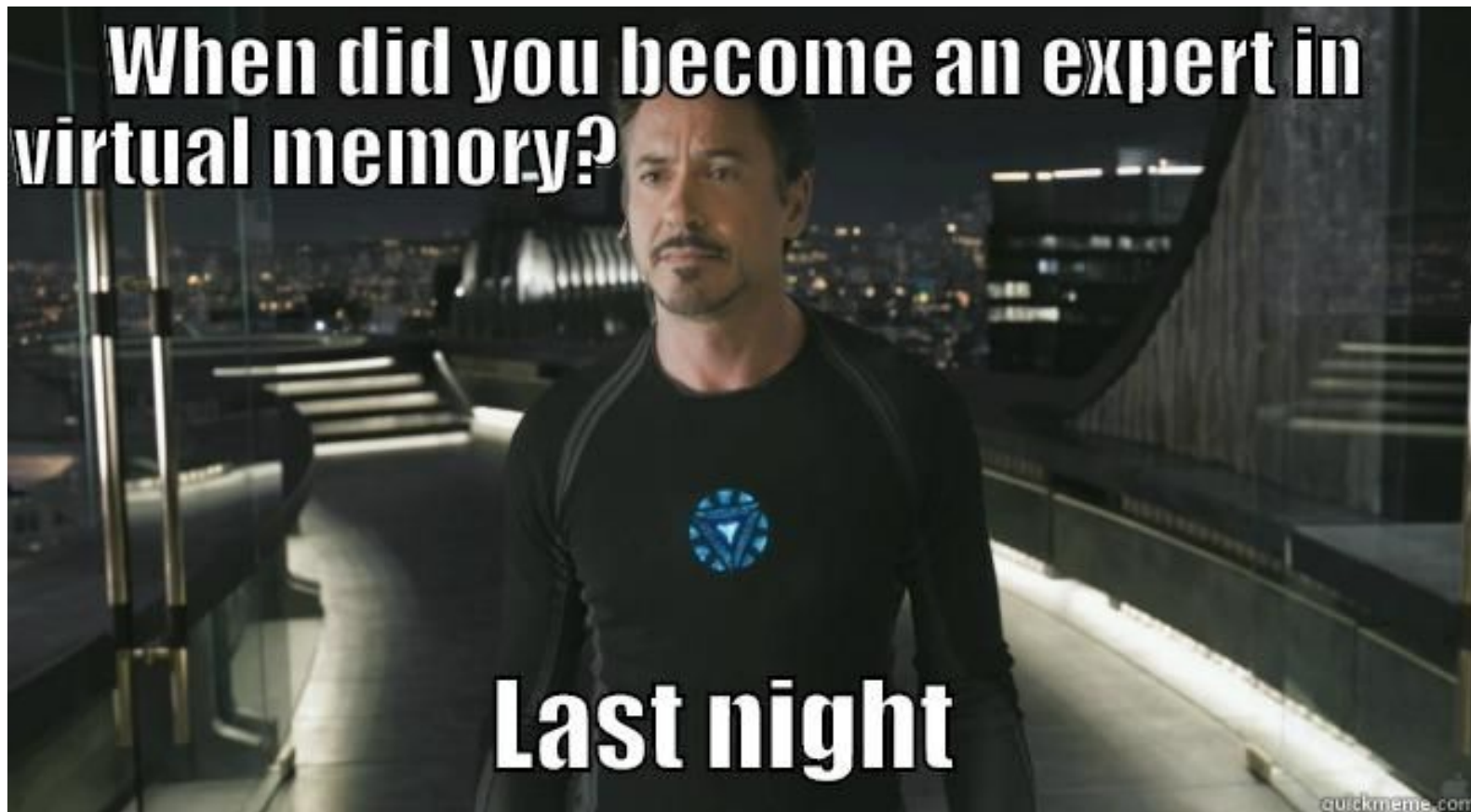


15-213: Final Exam Review

Kashish and Katherine

Virtual Memory



Virtual Memory

Virtual Address - 18 Bits

Physical Address - 12 Bits

Page Size - 512 Bytes

TLB is 8-way set associative

Cache is 2-way set associative

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
000	7	0	010	1	0
001	5	0	011	3	0
002	1	1	012	3	0
003	5	0	013	0	0
004	0	0	014	6	1
005	5	0	015	5	0
006	2	0	016	7	0
007	4	1	017	2	1
008	7	0	018	0	0
009	2	0	019	2	0
00A	3	0	01A	1	0
00B	0	0	01B	3	0
00C	0	0	01C	2	0
00D	3	0	01D	7	0
00E	4	0	01E	5	1
00F	7	1	01F	0	0

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
1	71	5	1
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

[Final S-02 \(#5\)](#)

[Lecture 18: VM - Systems](#)

Virtual Memory

Label the following:

- (A) *VPO*: Virtual Page Offset
- (B) *VPN*: Virtual Page Number
- (C) *TLBI*: TLB Index - Location in the TLB Cache
2 Indices \rightarrow 1 Bit

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

B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

TLBI

Virtual Memory

Label the following:

- (A) *VPO*: Virtual Page Offset
- (B) *VPN*: Virtual Page Number
- (C) *TLBI*: TLB Index
- (D) *TLBT*: TLB Tag - Everything Else

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

B B B B B B B B A A A A A A A A

TLBT

TLBI

Virtual Memory

Label the following:

- (A) *PPO*: Physical Page Offset - Same as VPO
- (B) *PPN*: Physical Page Number - Everything Else
- (C) *CO*: Cache Offset - Offset in Block

4 Byte Blocks \rightarrow 2 Bits



Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation

1 = 0001 A = 1010 9 = 1001 F = 1111 4 = 0100

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

Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation
2. Extract Information:

VPN: 0x?? TLBI: 0x?? TLBT: 0x??
TLB Hit: Y/N? Page Fault: Y/N? PPN: 0x??

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

Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation
2. Extract Information:

VPN: 0xD4 TLBI: 0x?? TLBT: 0x??
TLB Hit: Y/N? Page Fault: Y/N? PPN: 0x??

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

Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation
2. Extract Information:

VPN: 0xD4 TLBI: 0x00 TLBT: 0x??
TLB Hit: Y/N? Page Fault: Y/N? PPN: 0x??

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

Virtual Memory

Now to the actual question!

Q) Translate the following address: **0x1A9F4**

1. Write down bit representation
2. Extract Information:

VPN: 0xD4

TLBI: 0x00

TLBT: 0x6A

TLB Hit: Y/N?

Page Fault: Y/N?

PPN: 0x??

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
1	71	5	1
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

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

0	1	1	0	1	0	1	0	0	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation
2. Extract Information:

VPN: 0xD4 TLBI: 0x00 TLBT: 0x6A
 TLB Hit: Y! Page Fault: Y/N? PPN: 0x??

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
1	56	1	0
	60	4	1
	78	9	0
	71	5	1
1	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
	73	2	1

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

0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation
2. Extract Information:

VPN: 0xD4 TLBI: 0x00 TLBT: 0x6A
 TLB Hit: Y! Page Fault: N! PPN: 0x??

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
	1	71	5
31		A	1
53		F	0
87		8	0
51		D	0
39		E	1
43		B	0
73	2	1	

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

0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation
2. Extract Information:

VPN: 0xD4 TLBI: 0x00 TLBT: 0x6A

TLB Hit: Y! Page Fault: N! PPN: 0x3

TLB			
Index	Tag	PPN	Valid
0	55	6	0
	48	F	1
	00	A	0
	32	9	1
	6A	3	1
	56	1	0
	60	4	1
	78	9	0
1	71	5	1
	31	A	1
	53	F	0
	87	8	0
	51	D	0
	39	E	1
	43	B	0
73	2	1	

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

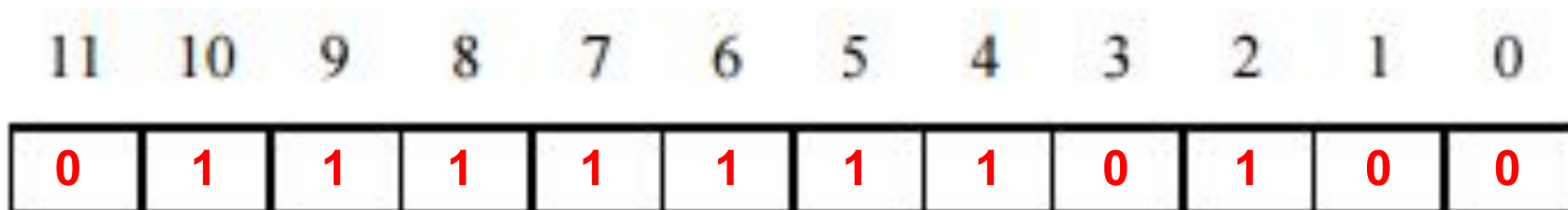
0	1	1	0	1	0	1	0	0	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory

Now to the actual question!

Q) Translate the following address: 0x1A9F4

1. Write down bit representation
2. Extract Information
3. Put it all together: PPN: 0x3, PPO = VPO = 0x1F4



Virtual Memory

Q) What is the value of the address?

CO: 0x?? CI: 0x?? CT: 0x?? Cache Hit: Y/N? Value:0x??

11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory

Q) **What is the value of the address?**

1. Extract more information

CO: 0x00 CI: 0x?? CT: 0x?? Cache Hit: Y/N? Value:0x??

11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	1	0	0

Virtual Memory

Q) What is the value of the address?

1. Extract more information

CO: 0x00 CI: 0x01 CT: 0x?? Cache Hit: Y/N? Value:0x??

11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	1	0	0

Virtual Memory

Q) What is the value of the address?

1. Extract more information
2. Go to Cache Table

CO: 0x00 CI: 0x01 CT: 0x7F Cache Hit: Y/N? Value:0x??

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory

Q) **What is the value of the address?**

1. Extract more information
2. Go to Cache Table

CO: 0x00 CI: 0x01 CT: 0x7F Cache Hit: Y Value:0x??

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory

Q) **What is the value of the address?**

1. Extract more information
2. Go to Cache Table

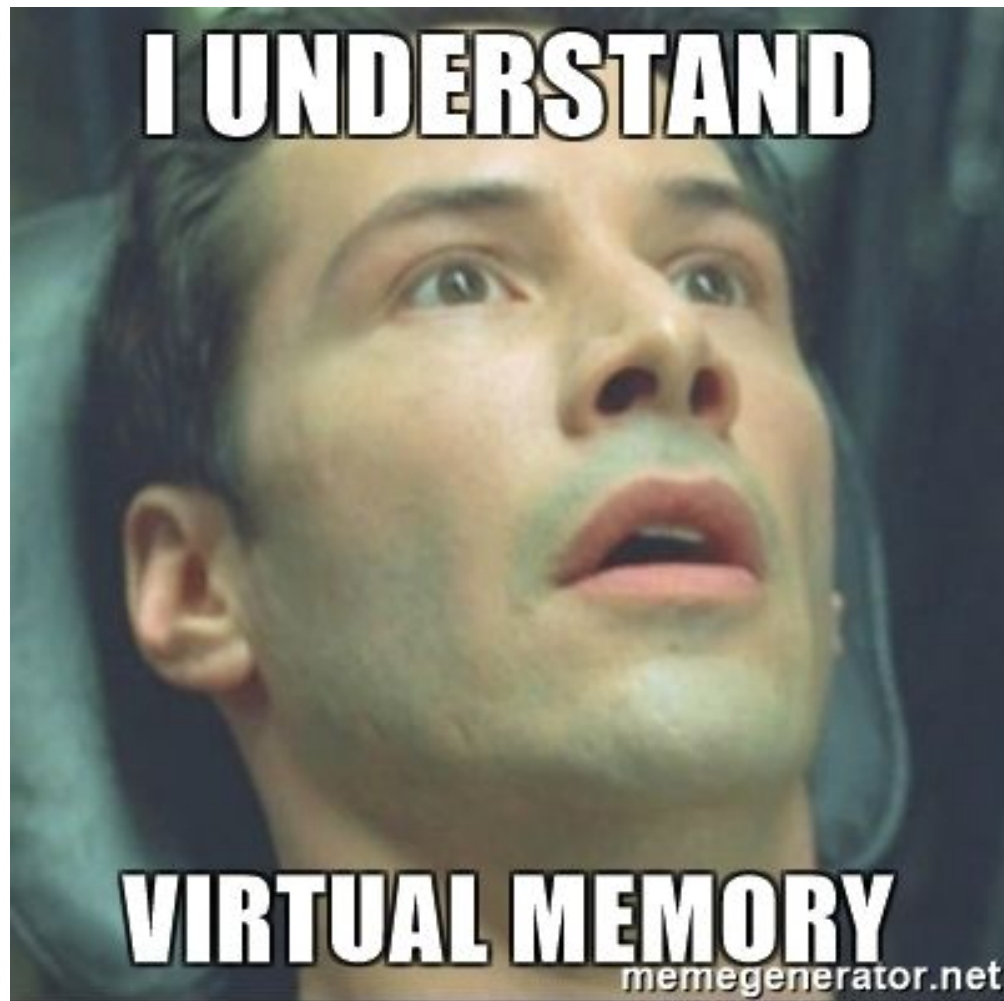
CO: 0x00 CI: 0x01 CT: 0x7F Cache Hit: Y Value:0xFF

2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

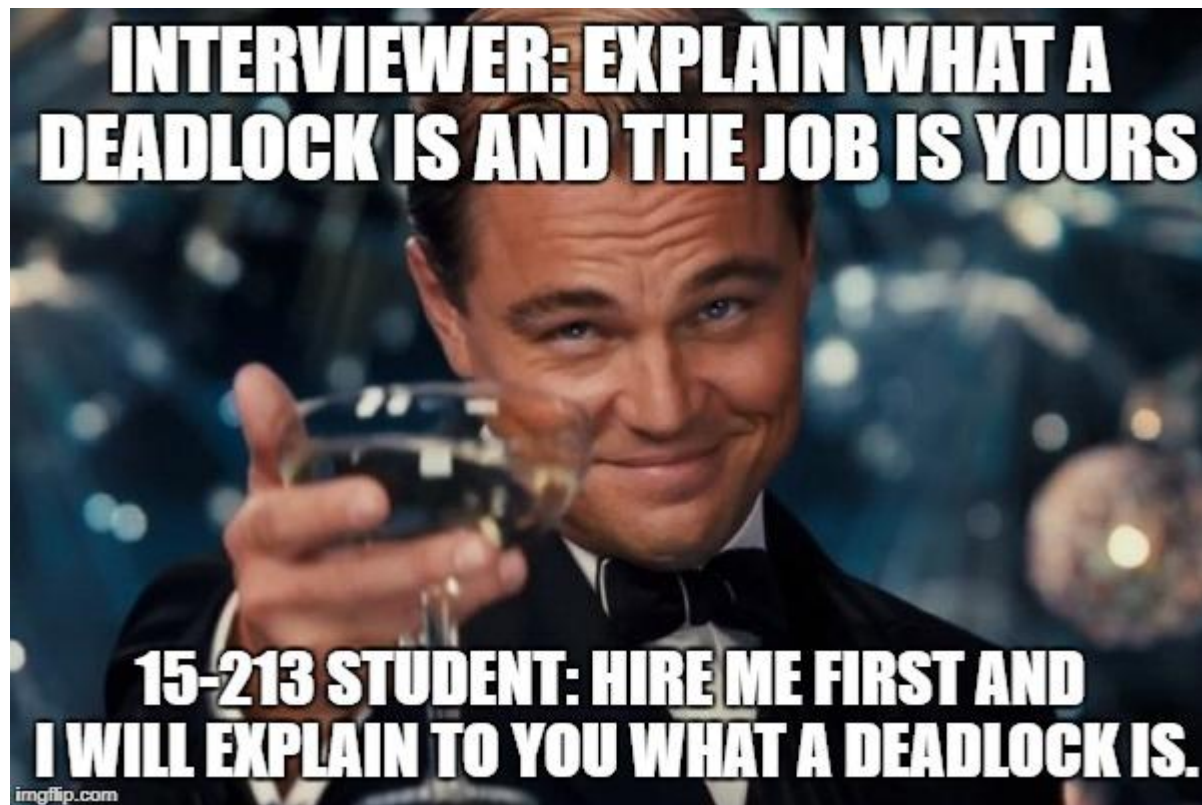
11 10 9 8 7 6 5 4 3 2 1 0

0	1	1	1	1	1	1	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---

Virtual Memory



Threads and Synchronization



Threads and Synchronization

Problem Statement:

- 15-213 TAs now want to begin a new procedure for daily office hours:
- When a student enters the room, he/she will see a table inside the room with several pens and several notebooks on it (if the pens and notebooks haven't been claimed by other students). TAs are inside the room waiting to help them.

Threads and Synchronization

Procedure for students:

- Wait until you get called, claim a pen and a notebook.
- Use the pen to write your questions on the notebook.
- After you are done with question writing, release the pen.
- Hold the notebook and wait until a TA is free to help you.
- After you are done with talking to a TA about the question, go back to claim a pen and write the solutions on the notebook.
- Release the pen and the notebook and go home.

Threads and Synchronization

Typical Code to launch threads and wait for them to finish:

```
sem_t pen;
sem_t notebook;
sem_t ta;

int main(int argc, char** argv)
{
    int P = 5, N = 5, T = 3, S = 50;
    sem_init(&pen, 0, P);
    sem_init(&notebook, 0, N);
    sem_init(&ta, 0, T);

    int* students = malloc(S * sizeof(int));
    launch_threads();
    reap_threads();
}
```

```
void launch_threads()
{
    for(int i = 0; i < S; i++)
    {
        students[s] =
            pthread_create(students+i,
                NULL, student_fn, NULL);
    }
}

void reap_threads()
{
    for(int i = 0; i < S; i++)
    {
        pthread_join(students[i]);
    }
}
```


Threads and Synchronization

A student writes the following worker thread to simulate the given procedure

```
void* student_fn(void* varp)
{
    P(&pen);
    P(&notebook);
    sleep(genrand() % 60); // Time for you to write questions

    V(&pen);
    P(&ta);
    sleep(genrand() % 900); // Time for you to ask TA

    V(&ta);
    P(&pen);
    sleep(genrand() % 60); // Time for you to write solutions

    V(&notebook);
    V(&pen);

    return 0;
}
```

- However, the code sometimes gets stuck by a deadlock. Which of the following is correct?
- A: The deadlock is caused by the pen and the notebook.
- B: The deadlock is caused by the notebook and the TA.
- C: The deadlock is caused by the TA and the pen.

Threads and Synchronization

A student writes the following program to simulate the given procedures

```
void* student_fn(void* varp)
{
    P(&pen);
    P(&notebook);
    sleep(genrand() % 60); // Time for you to write questions

    V(&pen);
    P(&ta);
    sleep(genrand() % 900); // Time for you to ask TA

    V(&ta);
    P(&pen);
    sleep(genrand() % 60); // Time for you to write solutions

    V(&notebook);
    V(&pen);

    return 0;
}
```

Let us consider a simple example:

- Pens: 5
- Notebooks: 5
- TAs:
- Students: 50

Threads and Synchronization

Let's analyse the semaphores

S: Signal void* student_fn(void* varp)

W: Wait {

W: Get a pen

```
P(&pen);
P(&notebook);
```

W: Get a Notebook

```
sleep(genrand() % 60); // Time for you to write questions
```

S: Give up pen

```
V(&pen);
```

W: Get a TA

```
P(&ta);
```

```
sleep(genrand() % 900); // Time for you to ask TA
```

S: Give up TA

```
V(&ta);
```

W: Get a pen

```
P(&pen);
```

```
sleep(genrand() % 60); // Time for you to write solutions
```

S: Give up Notebook

S: Give up pen

```
V(&notebook);
V(&pen);
```

```
return 0;
```

```
}
```

Case C: TAs and Pens

- There is no overlap
- There can't be a case where a student who is talking to a TA is waiting on a pen.
- There can't be a case where a student who is holding a pen is waiting on a TA

Threads and Synchronization

Let's analyse the semaphores

S: Signal void* student_fn(void* varp)

W: Wait {

W: Get a pen

```
P(&pen);
P(&notebook);
```

W: Get a Notebook

```
sleep(genrand() % 60); // Time for you to write questions
```

S: Give up pen

```
V(&pen);
```

W: Get a TA

```
P(&ta);
```

```
sleep(genrand() % 900); // Time for you to ask TA
```

S: Give up TA

```
V(&ta);
```

W: Get a pen

```
P(&pen);
```

```
sleep(genrand() % 60); // Time for you to write solutions
```

S: Give up Notebook

S: Give up pen

```
V(&notebook);
V(&pen);
```

```
return 0;
```

```
}
```

Case B: TAs and Notebooks

- There is an overlap
- A student who is holding a notebook may be waiting on a TA
- But....A student who is talking to a TA will not be waiting on a notebook

Threads and Synchronization

Let's analyse the semaphores

Case A: Pens and Notebooks

- There is an overlap

- A student who is holding a notebook may be waiting on a pen

- A student who is holding a pen may be waiting on a notebook

```

S: Signal      void* student_fn(void* varp)
W: Wait       {
W: Get a pen   P(&pen);
               P(&notebook);

               sleep(genrand() % 60); // Time for you to write questions

S: Give up pen V(&pen);
W: Get a TA    P(&ta);

               sleep(genrand() % 900); // Time for you to ask TA

S: Give up TA  V(&ta);
W: Get a pen   P(&pen);

               sleep(genrand() % 60); // Time for you to write solutions

               V(&notebook);
S: Give up pen V(&pen);

               return 0;
               }
  
```

W: Get a Notebook

S: Give up Notebook

Threads and Synchronization

Let's analyse the semaphores

S: Signal void* student_fn(void* varp)

W: Wait {

W: Get a pen

```
P(&pen);
P(&notebook);
```

W: Get a Notebook

```
sleep(genrand() % 60); // Time for you to write questions
```

S: Give up pen

```
V(&pen);
```

W: Get a TA

```
P(&ta);
```

```
sleep(genrand() % 900); // Time for you to ask TA
```

S: Give up TA

```
V(&ta);
```

W: Get a pen

```
P(&pen);
```

```
sleep(genrand() % 60); // Time for you to write solutions
```

S: Give up pen

```
V(&notebook);
V(&pen);
```

S: Give up Notebook

```
return 0;
```

```
}
```

Case A: Pens and Notebooks

Consider this likely scenario:

- 5 Students (group A) enter the room
- They grab all 5 pens and all 5 notebooks on the table.
- Every student behind them is waiting on a pen and a notebook. (Group B)
- All 5 students in Group A give up their pens, but not their notebooks.
- 5 students in Group B immediately grab the 5 pens.
- Group B are now waiting on notebooks.
- After talking to the TAs, all students in Group A are now waiting on pens, which students in Group B have.
- Group B is waiting on the notebooks which students in Group A have.

DEADLOCK!!

Threads and Synchronization

A student writes the following program to simulate the given procedures

```
void* student_fn(void* varp)
{
    P(&pen);
    P(&notebook);

    sleep(genrand() % 60); // Time for you to write questions

    V(&pen);
    P(&ta);

    sleep(genrand() % 900); // Time for you to ask TA

    V(&ta);
    P(&pen);

    sleep(genrand() % 60); // Time for you to write solutions

    V(&notebook);
    V(&pen);

    return 0;
}
```

- However, the code sometimes gets stuck by a deadlock. Which of the following is correct?
- A: The deadlock is caused by the pen and the notebook.
- B: The deadlock is caused by the notebook and the ta.
- C: The deadlock is caused by the ta and the pen.

ANSWER: A

Threads and Synchronization

Can we fix this without changing the Office Hour procedure?

Threads and Synchronization

Can we fix this without changing the Office Hour procedure?

YES!

Threads and Synchronization

Let's analyse the semaphores

```
void* student_fn(void* varp)
{
    P(&notebook);
    P(&pen);

    sleep(genrand() % 60); // Time for you to write questions

    V(&pen);
    P(&ta);

    sleep(genrand() % 900); // Time for you to ask TA

    V(&ta);
    P(&pen);

    sleep(genrand() % 60); // Time for you to write solutions

    V(&pen);
    V(&notebook);

    return 0;
}
```

W: Get a pen

S: Give up pen

W: Get a TA

S: Give up TA

W: Get a pen

S: Give up pen

S: Signal
W: Wait

W: Get a Notebook

S: Give up Notebook

Let's just re-order the locks and analyse it again.

- A student who is holding a notebook may wait on a pen or a TA

However...

- A student who is holding a pen will already have a notebook (and never wait on it).
- A student who is talking to a TA will already have a notebook (and never wait on it).

Processes



Processes

Draw a Process Graph!!!

(it does not have to be like mine)

Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

What is printed?

Assume printf is atomic,
and all system calls
succeed.

Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

How many processes?

Processes

```
int main() {  
    int count = 1;  
    int pid1 = fork();  
    int pid2 = fork();  
  
    if(pid1 == 0)  
        count++;  
    else{  
        if(pid2 == 0)  
            count--;  
        else  
            count += 2;  
    }  
    printf("%d", count);  
}
```

How many processes?

Parent: forks child

Parent and child: each fork
another child

Total: 4 processes

Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

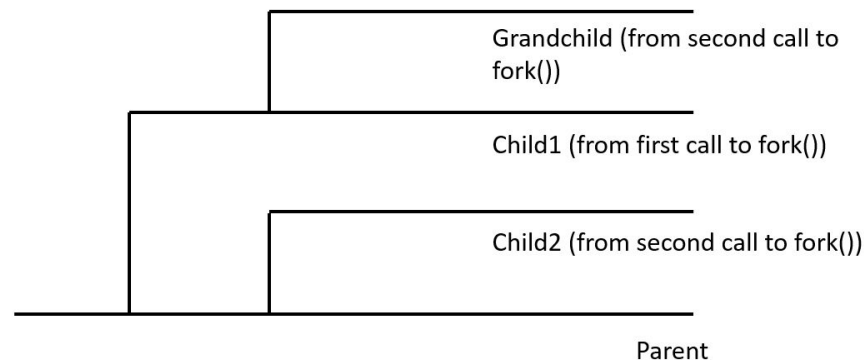
What does the process diagram look like?

Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

What does the process diagram look like?



Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

What does count look like?

Parent: pid1 != 0 and pid2 != 0

Child1: pid1 == 0 and pid2 != 0

Child2: pid1 != 0 and pid2 == 0

Grandchild: pid1 == 0 and pid2 == 0

Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

What does count look like?

Parent: pid1 != 0 and pid2 != 0

- count = 3

Child1: pid1 == 0 and pid2 != 0

- count = 2

Child2: pid1 != 0 and pid2 == 0

- count = 0

Grandchild: pid1 == 0 and pid2 == 0

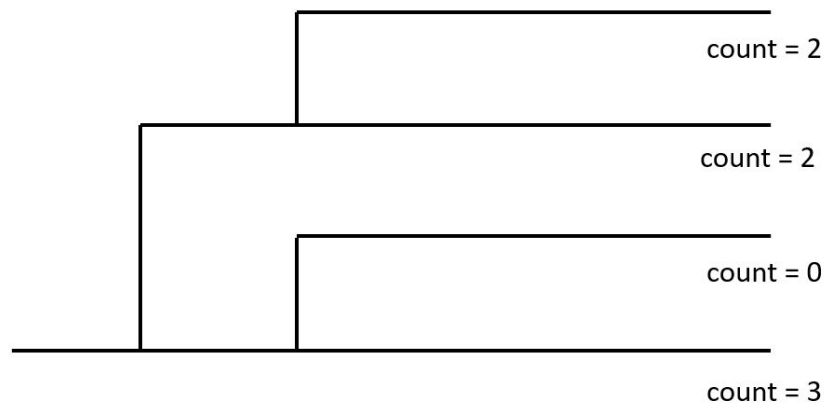
- count = 2

Processes

```
int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}
```

Given the process diagram, what are the different permutations that can be printed out?



Processes

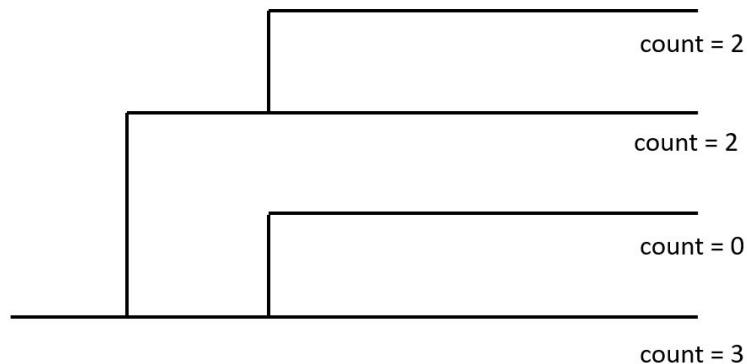
```

int main() {
    int count = 1;
    int pid1 = fork();
    int pid2 = fork();

    if(pid1 == 0)
        count++;
    else{
        if(pid2 == 0)
            count--;
        else
            count += 2;
    }
    printf("%d", count);
}

```

Given the process diagram, what are the different permutations that can be printed out?



Math! $4! / 2 = 12$ different possible outcomes

Processes



Remember:

- Processes can occur in any order
- Watch out for a `wait` or a `waitpid`!
 - What if I included a `wait(NULL)` before I printed out count?
- Good luck!

File IO



File IO

```
foo.txt: abcdefgh...xyz
int main() {
    int fd1, fd2, fd3;
    char c;
    pid_t pid;
    fd1 = open("foo.txt", O_RDONLY);
    fd2 = open("foo.txt", O_RDONLY);
    fd3 = open("foo.txt", O_RDONLY);
    read(fd1, &c, sizeof(c)); // c = ?
    read(fd2, &c, sizeof(c)); // c = ?
    dup2(fd2, fd3);
    read(fd3, &c, sizeof(c)); // c = ?
    read(fd2, &c, sizeof(c)); // c = ?
}
```

Main ideas:

- How does read offset?
- How does dup2 work?
 - What is the order of arguments?
 - Does fd3 share offset with fd2?

File IO

```
foo.txt: abcdefgh...xyz
int main() {
    int fd1, fd2, fd3;
    char c;
    pid_t pid;
    fd1 = open("foo.txt", O_RDONLY);
    fd2 = open("foo.txt", O_RDONLY);
    fd3 = open("foo.txt", O_RDONLY);
    read(fd1, &c, sizeof(c)); // c = a
    read(fd2, &c, sizeof(c)); // c = a
    dup2(fd2, fd3);
    read(fd3, &c, sizeof(c)); // c = b
    read(fd2, &c, sizeof(c)); // c = c
}
```

- How does read offset?
 - Incremented by number of bytes read
- How does dup2 work?
 - Any read/write from fd3 now happen from fd2
 - All file offsets are shared

File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

Main ideas:

- How are fd shared between processes?
- How does dup2 work from parent to child?
- How are file offsets shared between processes?

File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c
```

```
pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

What would this program print?

File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

Possible output 1:

```
c = d // in parent
c = b // in parent
c = c // in child from fd1
c = e // in child from fd3
c = d // in child
c = e // in child
```

File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
```

Possible output 2:

```
c = b // in child
c = d // in child
c = c // in child
c = d // in child
c = e // in parent
c = e // in parent
```

File IO

```
.  
.br/>pid = fork();  
if (pid==0) {  
    read(fd1, &c, sizeof(c));  
    printf("c = %c\n", c);  
    dup2(fd1, fd2);  
    read(fd3, &c, sizeof(c));  
    printf("c = %c\n", c);  
}  
if (pid!=0) waitpid(-1, NULL, 0);  
read(fd2, &c, sizeof(c));  
printf("c = %c\n", c);  
read(fd1, &c, sizeof(c));  
printf("c = %c\n", c);  
return 0;  
}
```

What are the possible outputs now?

File IO

```
.  
.
pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    printf("c = %c\n", c);
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
    printf("c = %c\n", c);
}
if (pid!=0) waitpid(-1, NULL, 0);
read(fd1, &c, sizeof(c));
printf("c = %c\n", c);
read(fd2, &c, sizeof(c));
printf("c = %c\n", c);
return 0;
}
```

Possible output:

c = b // in child

c = d // in child

c = c // in child

c = d // in child

c = e // in parent

c = e // in parent

File IO

```
read(fd1, &c, sizeof(c)); // a
read(fd2, &c, sizeof(c)); // a
dup2(fd2, fd3);
read(fd3, &c, sizeof(c)); // b
read(fd2, &c, sizeof(c)); // c

pid = fork();
if (pid==0) {
    read(fd1, &c, sizeof(c));
    dup2(fd1, fd2);
    read(fd3, &c, sizeof(c));
}
if (pid!=0) waitpid(-1, NULL, 0);
read(fd2, &c, sizeof(c));
read(fd1, &c, sizeof(c));
```

- Child creates a copy of the parent fd table
 - dup2/open/close in parent affect the child
 - dup2/open/close in child do NOT affect the parent
- File descriptors across process share the same file offset.

Good luck!



Signals

who would win?

several hundred lines of
tshlab code

```
7 void
6 exec_cmdline(char *cmdline, char **argv, sigset_t *set,
5             int bg, int fd_in, int fd_out)
4 {
3     pid_t pid = 0;
2     if ((pid = Fork()) == 0) {
1         // Child process; restore mask and execute job.
314 |         Sigprocmask(SIG_SETMASK, set, NULL);
1
```

one asynchronous boi



SIGCHLD

Signals

- Examples in these slides are important “gotchas”
 - **NOT** an inclusive list
 - In particular, these do **NOT** go over signals sent **between multiple processes**
- Many more examples in:
 - Recitation slides
 - Lecture slides
 - Past exams
- Reminder: solve the questions **on your own** when studying **before** looking at the answer

Signals

- Does the following code ever terminate?

```
void handler(int sig) {  
    while(1);  
}  
  
int main() {  
    Signal(SIGUSR1, handler);  
    Kill(0, SIGUSR1);  
    return 0;  
}
```

Signals

- Does the following code ever terminate?

```
void handler(int sig) {
    while(1);           // stuck here forever!
}

int main() {
    Signal(SIGUSR1, handler);
    // Spec says signal sent to self must be handled
    // before kill() returns
    Kill(0, SIGUSR1);
    return 0;
}
```

Signals

- What about now?

```
void handler(int sig) {  
    Kill(0, SIGUSR1);  
}
```

```
int main() {  
    Signal(SIGUSR1, handler);  
    Kill(0, SIGUSR1);  
    return 0;  
}
```

Signals

- What about now?

```
void handler(int sig) {
    Kill(0, SIGUSR1);
}

int main() {
    Signal(SIGUSR1, handler);
    Kill(0, SIGUSR1);
    return 0;
}
```

- Does not terminate:

```
main: Kill(0, SIGUSR1);
<SIGUSR1 handler invoked,
SIGUSR1 blocked in handler>
handler: Kill(0, SIGUSR1);
<SIGUSR1 pending>
<handler returns>
<SIGUSR1 unblocked, SIGUSR1
handler invoked, SIGUSR1
blocked in handler>
handler: Kill(0, SIGUSR);
...repeat...
```

Malloc



Malloc

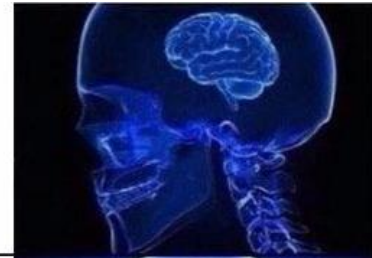
- Fit algorithms - first/next/best/good
- Fragmentation
 - Internal - inside blocks
 - External - between blocks
- Organization
 - Implicit
 - Explicit
 - Segregated

GOOD FIT

BEST FIT

FIRST FIT

EXTEND
HEAP



Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)						
b = malloc(16)						
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)						
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)						
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)						
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)						

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
 - internal?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
 - internal
 - $(48-16) + (80-48) = 64$
 - external?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

Malloc - First fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
 - internal
 - $(48-16) + (80-48) = 64$
 - external
 - 32

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48a	32a	32f [0]	48a		
free(d)	48a	32a	80f [0]			
f = malloc(48)	48a	32a	80a			
free(b)	48a	32f [0]	80a			

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)						
b = malloc(16)						
c = malloc(16)						
d = malloc(40)						
free(c)						
free(a)						
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)						
free(d)						
f = malloc(48)						
free(b)						

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)						
f = malloc(48)						
free(b)						

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)						
free(b)						

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	54a		
free(b)						

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	54a		
free(b)	48f [1]	32f [0]	32a	54a		

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
 - internal?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	54a		
free(b)	48f [1]	32f [0]	32a	54a		

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
 - internal
 - $(32-16) + (54-48) = 24$
 - external?

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	54a		
free(b)	48f [1]	32f [0]	32a	54a		

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
 - internal
 - $(32-16) + (54-48) = 24$
 - external
 - $48+32 = 80$

	#1	#2	#3	#4	#5	#6
a = malloc(32)	48a					
b = malloc(16)	48a	32a				
c = malloc(16)	48a	32a	32a			
d = malloc(40)	48a	32a	32a	48a		
free(c)	48a	32a	32f [0]	48a		
free(a)	48f [0]	32a	32f [1]	48a		
e = malloc(16)	48f [0]	32a	32a	48a		
free(d)	48f [1]	32a	32a	48f [0]		
f = malloc(48)	48f [0]	32a	32a	54a		
free(b)	48f [1]	32f [0]	32a	54a		

Good luck!

