15-213/513/613: Final Exam Review

Xuwen, Chaojiang, Yanwen

Final Exam Logistics

- Physical Cheat sheets - 2 double sided 8.5 x 11 in.
- There will be two in-person exam sessions:
- 1.Thursday, August 11 at 12:20pm in Rashid Auditorium
- 2.Saturday, August 27 at 12:00pm in Rashid Auditorium
- Details can be found at piazza note @2345

- Cumulative:
  - Floats, Assembly, Structs, Stack, Caching, Malloc, VM, Processes, Signals, IO, Threads

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

Final S-02 (#5)

Lecture 18: VM - Systems

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

Virtual Memory

Label the following:

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

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

Virtual Memory

Label the following:

(A) *VPO:* Virtual Page Offset - Location in the page

Page Size = 512 Bytes = 2^9 → Need 9 bits

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

Virtual Memory

Label the following:

(A) *VPO:* Virtual Page Offset

(B) *VPN:* Virtual Page Number - 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	B	A	A	A	A	A	A	A	A	A

Virtual Memory

Label the following:

(A) *VPO:* Virtual Page Offset

(B) *VPN:* Virtual Page Number

(C) *TLBI:* TLB Index - Location in the TLB Cache

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

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 → 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

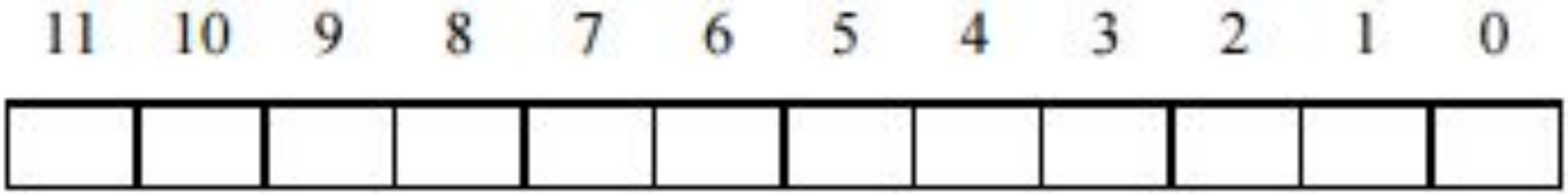
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A	A

TLBT TLBI

Virtual Memory

Label the following:

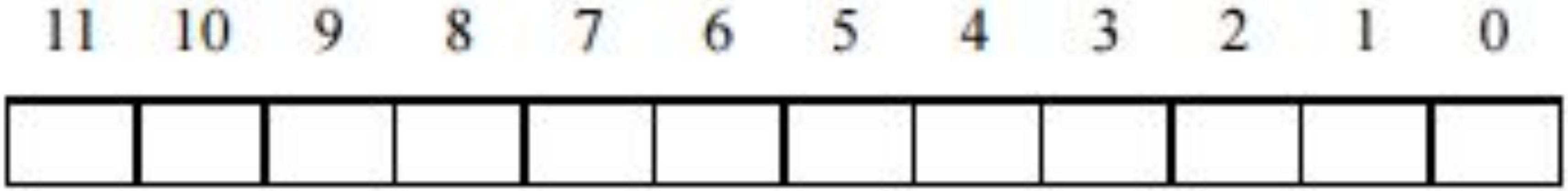
(A) *PPO:* Physical Page Offset
(B) *PPN:* Physical Page Number
(C) *CO:* Cache Offset
(D) *CI:* Cache Index
(E) *CT:* Cache Tag



Virtual Memory

Label the following:

(A) *PPO:* Physical Page Offset



Virtual Memory

Label the following:

(A) *PPO:* Physical Page Offset - Same as VPO

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

Virtual Memory

Label the following:

(A) *PPO:* Physical Page Offset - Same as VPO

(B) *PPN:* Physical Page Number - Everything Else

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

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

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

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 → 2 Bits

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

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

(D) *CI:* Cache Index

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

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

(D) *CI:* Cache Index

4 Indices → 2 Bits

11	10	9	8	7	6	5	4	3	2	1	0
B	B	B	A	A	A	A	A	A	A	A	A
								CI		CO	

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

(D) *CI:* Cache Index

(E) *CT:* Cache Tag - Everything Else

11	10	9	8	7	6	5	4	3	2	1	0
B	B	B	A	A	A	A	A	A	A	A	A
Cache Tag								CI		CO	

Virtual Memory

Now to the actual question!

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

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

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

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

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??

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: 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: 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	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 = 0x??

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

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

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?**

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?**

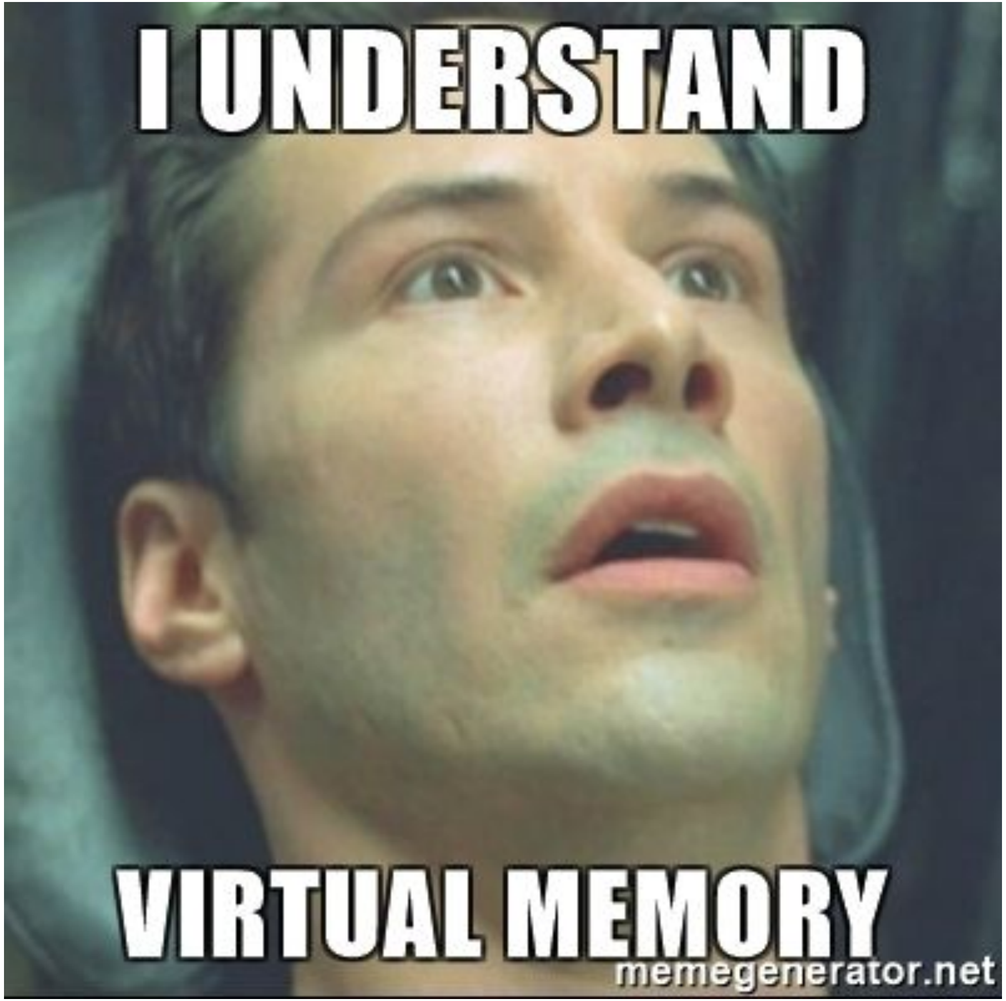
1. Extract more information
2. Go to Cache Table

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

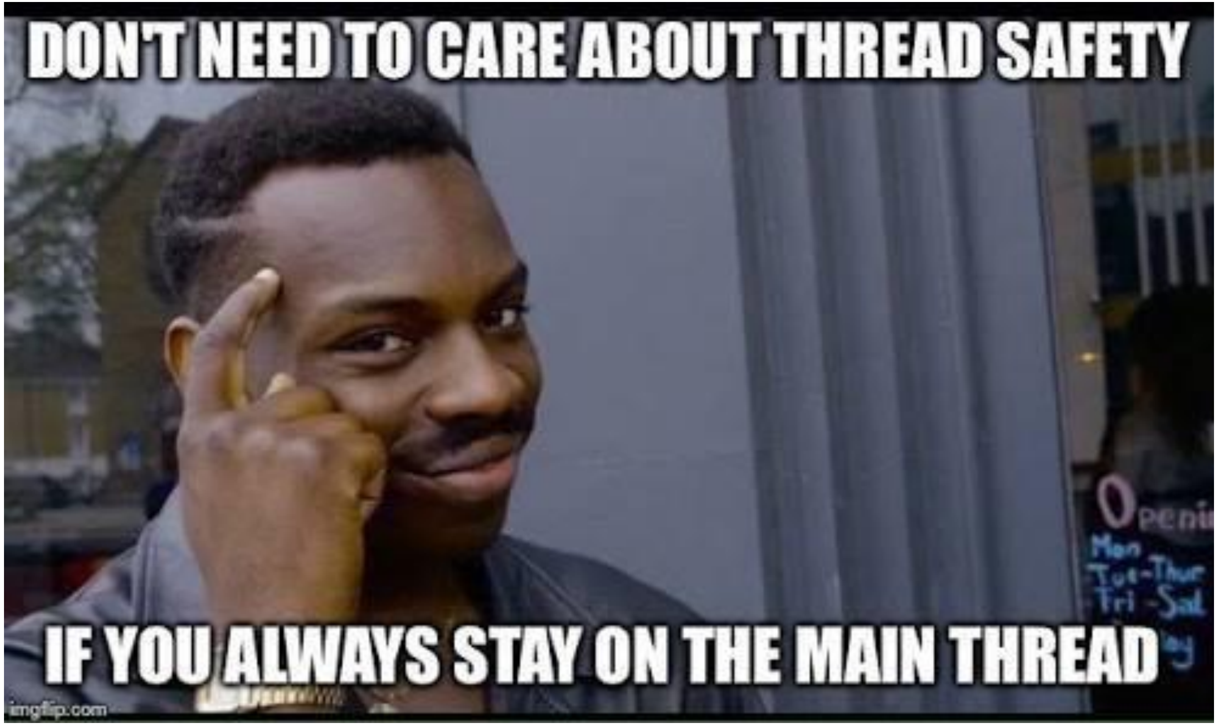
2-way Set Associative Cache												
Index	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3	Tag	Valid	Byte 0	Byte 1	Byte 2	Byte 3
0	7A	1	09	EE	12	64	00	0	99	04	03	48
1	02	0	60	17	18	19	7F	1	FF	BC	0B	37
2	55	1	30	EB	C2	0D	0B	0	8F	E2	05	BD
3	07	1	03	04	05	06	5D	1	7A	08	03	22

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


DON'T NEED TO CARE ABOUT THREAD SAFETY
IF YOU ALWAYS STAY ON THE MAIN THREAD
imgflip.com

Threads

Given this code, what variables do you think are shared?

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 2
int balance = 10;


 int main() {
     int i;
     pthread_t tid[NUM_THREADS];
     pthread_create(&tid[0], NULL, threadA, (void*)0);
     pthread_create(&tid[1], NULL, threadB, (void*)0);
     for (i = 0; i < NUM_THREADS; i++) {
         pthread_join(tid[i], NULL);
     }
     printf("balance: %d\n", balance); // What is balance?
     return 0;
 }
```

```c
void *threadA(void *vargp) {
     long instance = (long)vargp;
     static int cnt = 0;
     deposit(4);
     withdraw(11);
     return NULL;
}

void *threadB() {
     withdraw(6);
     deposit(3);
     withdraw(7);
     return NULL;
}
```

Threads (Contd.)

Which variables can be shared by multiple threads simultaneously in this program?

(A) i
(B) balance
(C) instance
(D) cnt
(E) None of the above

Threads (Contd.)

Which variables can be shared by multiple threads simultaneously in this program?

(A) i
(B) balance
(C) instance
(D) cnt
(E) None of the above

Answer: B

Threads (Contd.)

(A) i is a local variable so it isn’t shared.

(A) balance is a global variable so it’s shared.

(A) instance is local to threadA() so it isn’t shared.

(A) cnt is a static variable, so it retains its value even outside the scope in which it was defined, so it isn’t shared.

Threads (Contd.)

Given the withdraw() and deposit() functions, what are the possible outputs? (balance = 10 initially)

```
int withdraw(int amt) {
    if (balance >= amt) {
        balance = balance - amt;
        return 0;
    } else {
        return -1;
    }

int deposit(int amt) {
    balance = balance + amt;
    sleep(2);
    return 0;
}
```

```
void *threadA(void *vargp) {
    long instance = (long)vargp;
    static int cnt = 0;
    deposit(4);
    withdraw(11);
    return NULL;
}

void *threadB() {
    withdraw(6);
    deposit(3);
    withdraw(7);
    return NULL;
}
```

Threads (Contd.)

What can be the value of balance?

(A) balance: 0
(B) balance: -3
(C) balance: 14
(D) balance: 6
(E) balance: 17
(F) balance: 4

Threads (Contd.)

What can be printed at the indicated line?

(A) balance: 0
(B) balance: -3
(C) balance: 14
(D) balance: 6
(E) balance: 17
(F) balance: 4

Answer: ABDF

Threads (Contd.)

The following is one interleaving that leads to output 0:

- Thread A executes deposit(4), balance = 14
- Thread B executes withdraw(6), balance = 8
- Thread B executes deposit(3), balance = 11
- Thread A executes withdraw(11), balance = 0
- Thread B executes withdraw(7), balance = 0

Threads (Contd.)

The following is one interleaving that leads to output -3:

- Thread A executes deposit(4), balance = 14
- Thread A starts to execute withdraw(11) and enters the if condition
- Thread B executes withdraw(6), balance = 8
- Thread A computes RHS for withdraw(11) = -3
- Thread B executes deposit(3), balance = 11
- Thread A completes withdraw(11), balance = -3
- Thread B executes withdraw(7), balance = -3

Threads (Contd.)

The following is one interleaving that leads to output 6:

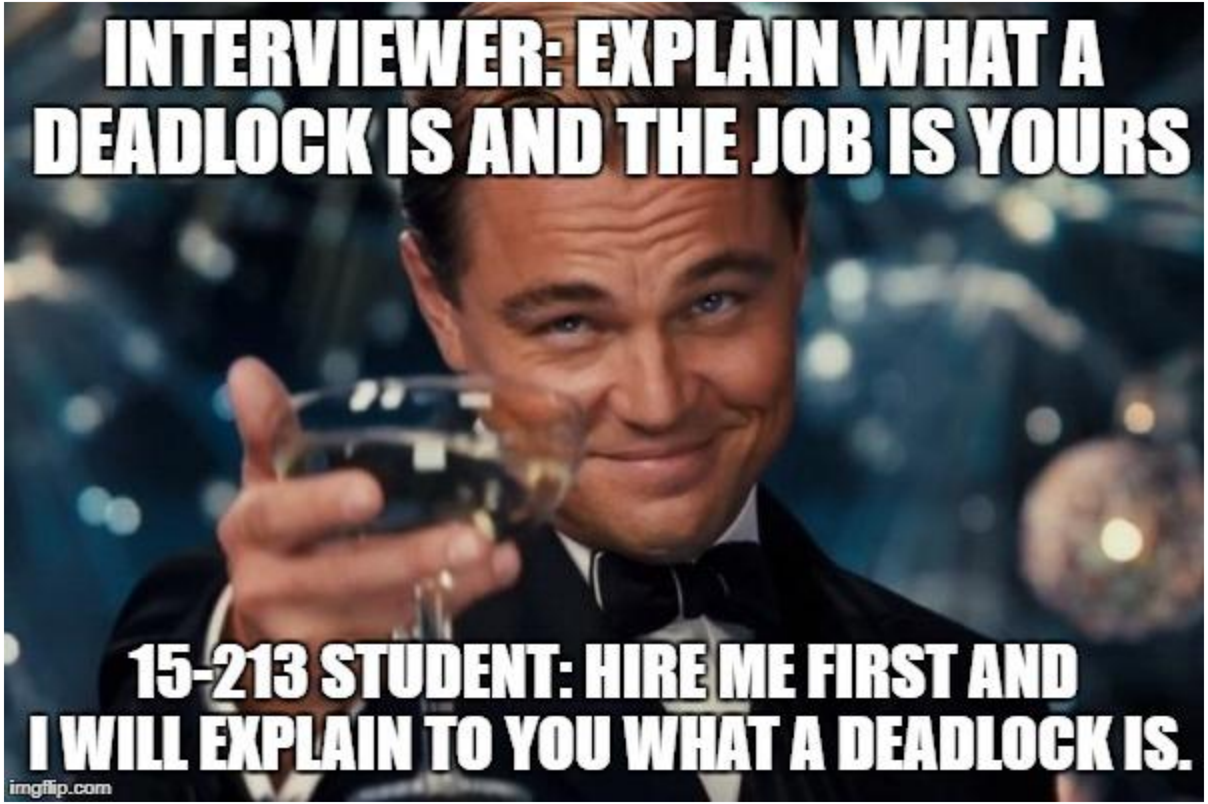
- Thread A executes deposit(4), balance = 14
- Thread A executes withdraw(11), balance = 3
- Thread B executes withdraw(6), balance = 3
- Thread B executes deposit(3), balance = 6
- Thread B executes withdraw(7), balance = 6

Threads (Contd.)

The following is one interleaving that leads to output 4:

- Thread B executes withdraw(6), balance = 4
- Thread A executes deposit(4), balance = 8
- Thread A executes withdraw(11), balance = 8
- Thread B executes deposit(3), balance = 11
- Thread B executes withdraw(7), balance = 4

Synchronization



Thread Synchronization

How many potential deadlock situations are present?

```
void *thread1(void *vargp) {
    V(&add_sem);
    V(&rem_sem);

    remove();

    P(&add_sem);
    P(&rem_sem);

    add();

    V(&add_sem);
    V(&rem_sem);

    remove();
    add();
}

sem_t add_sem;
sem_t rem_sem;
```

```
void *thread2(void *vargp) {
    P(&rem_sem);
    P(&add_sem);

    add();
    remove();
}

int main() {
    pthread_t tid1, tid2;

    sem_init(&add_sem,0,0);
    sem_init(&rem_sem,0,0);

    pthread_create(&tid1, NULL, thread1, NULL);
    pthread_create(&tid2, NULL, thread2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

Thread Synchronization (Contd.)

Situation 1:

tid1 executes V(&add_sem) and V(&rem_sem). Then, tid2 executes P(&rem_sem) and P(&add_sem). In this situation, tid1 can never execute P(&add_sem) since the value of add_sem = 0. As a result, this is a deadlock, since after the execution of thread 2, thread 1 can't resume. Thus, there's a deadlock.

Thread Synchronization (Contd.)

Situation 2:

tid1 executes V(&add_sem) and V(&rem_sem). Then, tid2 executes P(&rem_sem). Next, tid1 executes P(&add_sem). Thread 2 wants to execute P(&add_sem) but it can’t since add_sem has value 0. Thread 1 wants to execute P(&rem_sem) but it can’t since rem_sem has value 0. Thus, there’s a deadlock.

Thread Synchronization (Contd.)

For lengths 0-6, indicate the number of outcomes of that length that can be produced.

```c
sem_t add_sem;
sem_t rem_sem;

void add() {
    printf("A");
}

void remove() {
    printf("R");
}

void *thread2(void *vargp)
    P(&rem_sem);
    P(&add_sem);

    add();
    remove();
}
```

```c
void *thread1(void *vargp) {
    V(&add_sem);
    V(&rem_sem);

    remove();

    P(&add_sem);
    P(&rem_sem);

    add();

    V(&add_sem);
    V(&rem_sem);

    remove();
    add();
}
```

```c
int main() {
    pthread_t tid1, tid2;

    sem_init(&add_sem,0,0);
    sem_init(&rem_sem,0,0);

    pthread_create(&tid1, NULL, thread1, NULL);
    pthread_create(&tid2, NULL, thread2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

Thread Synchronization (Contd.)

Response length 0: None

This is because at least ‘R’ must get printed due to the call to remove() in thread1(). Even if there is a deadlock, at least that statement gets executed by tid1 before any sort of deadlock from the above situations.

Thread Synchronization (Contd.)

Response length 1: 1 (R)

In the deadlock scenario 2, where thread 1 executes P(&add_sem) and thread 2 executes P(&rem_sem), neither of the threads can proceed past that. Thus, no print statements are executed in either thread after that point. The only print statement that gets executed is due to the call to remove() before the calls to P() in thread 1.

Thread Synchronization (Contd.)

Response length 2: None

We noticed that ‘R’ due to the call to remove() in thread1() gets printed no matter what. From the code, we notice that it’s not possible for only one other print statement to get executed.

Thread Synchronization (Contd.)

Response length 3: 2 (RAR, ARR)

This happens due to deadlock scenario 1 above, where thread2() executes completely but thread1() can’t execute P(&add_sem) and the statements after that.

- RAR: Thread 1 executes remove(), followed by thread 2 executing add() and remove().
- ARR: Thread 2 executes add(), followed by any ordering of the 2 calls to remove() by threads 1 and 2.

Thread Synchronization (Contd.)

Response length 4: None

For any length greater than 3, it means that there was no deadlock, since thread 2 could run to completion and thread 1 could get past the calls to P(), which means it would run to completion as well. Thus, no responses of length greater than 3 and less than 6 are possible.

Thread Synchronization (Contd.)

Response length 5: None

For any length greater than 3, it means that there was no deadlock, since thread 2 could run to completion and thread 1 could get past the calls to P(), which means it would run to completion as well. Thus, no responses of length greater than 3 and less than 6 are possible.

Thread Synchronization (Contd.)

Response length 6: 4
(RARAAR, RARARA, RAARRA, RAARAR)

Since there are no deadlocks, it means that the initial calls to V() and P() get executed by thread 1. Thus, ‘R’ and ‘A’ definitely get printed. After this, the calls to V() get executed by thread 1 and then, thread 2 can execute its calls to P(). After this, based on the interleavings between the threads, there are 4 possible outputs.

Thread Synchronization (Contd.)

- RARAAR: Thread 1 executes remove(), threads 1 and 2 execute the add() statements in any order, and then thread 2 executes remove().

- RARARA: Thread 1 executes remove(), thread 2 executes add() and remove(), then thread 1 executes add().

Thread Synchronization (Contd.)

- RAARRA: Thread 2 executes add(), threads 1 and 2 execute the remove() statements in any order, and then thread 1 executes add().

- RAARAR: Thread 2 executes add(), thread 1 executes remove() and add(), then thread 2 executes remove().

Good luck!

Processes

1. Logical control flow
2. Private address space

Important system calls

1. Fork
2. Execve
3. Wait
4. Waitpid

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);
}
```

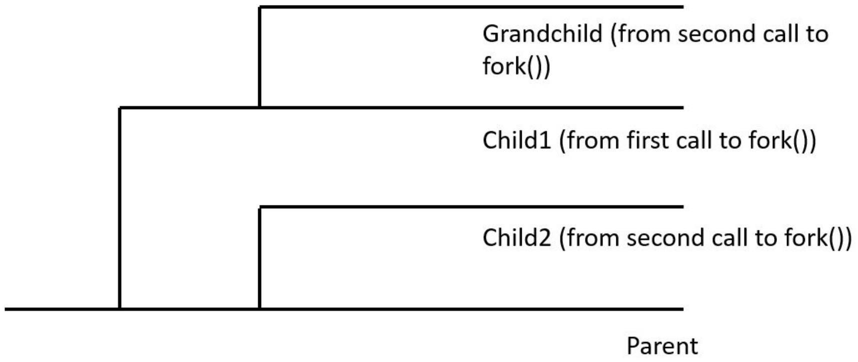
What does the process diagram look like?

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

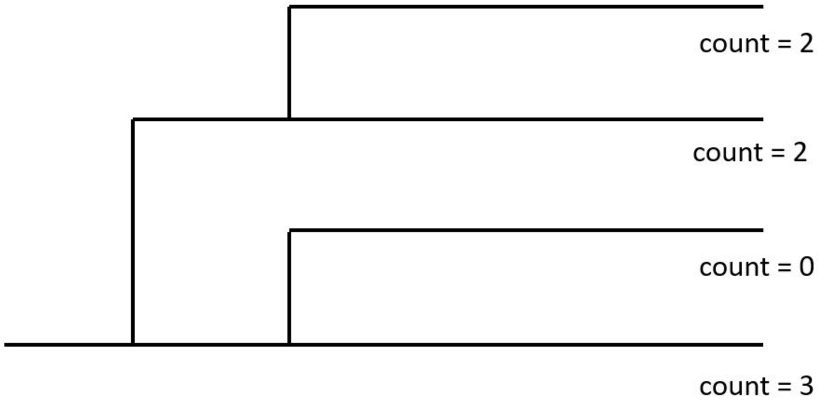
Grandchild: pid1 == 0 and pid2 == 0
- count = 2

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?

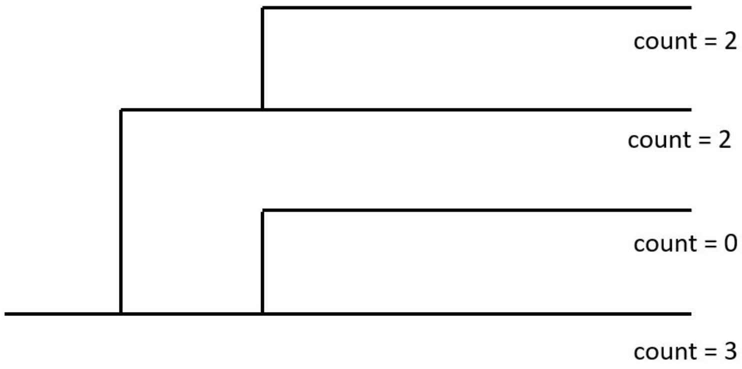


Processes

```
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!

Signals

who would win?

several hundred lines of tshlab code

```c
void
exec_cmdline(char *cmdline, char **argv, sigset_t *set,
        int bg, int fd_in, int fd_out)
{
    pid_t pid = 0;
    if ((pid = Fork()) == 0) {
        // Child process; restore mask and execute job.
        Sigprocmask(SIG_SETMASK, set, NULL);
```

one asynchronous boi

SIGCHLD

Signals

- **Child calls kill(parent, SIGUSR{1,2}) between 2-4 times. What sequence of kills may print 1? Can you guarantee printing 2? What is the range of values printed?**

```
int counter = 0;
void handler (int sig) {
  atomically {counter++;}
}
int main(int argc, char** argv) {
  signal(SIGUSR1, handler);
  signal(SIGUSR2, handler);
  int parent = getpid();   int child = fork();
  if (child == 0) {
    /* insert code here */
    exit(0);
  }
  sleep(1);   waitpid(child, NULL, 0);
  printf("Received %d USR{1,2} signals\n", counter);
}
```

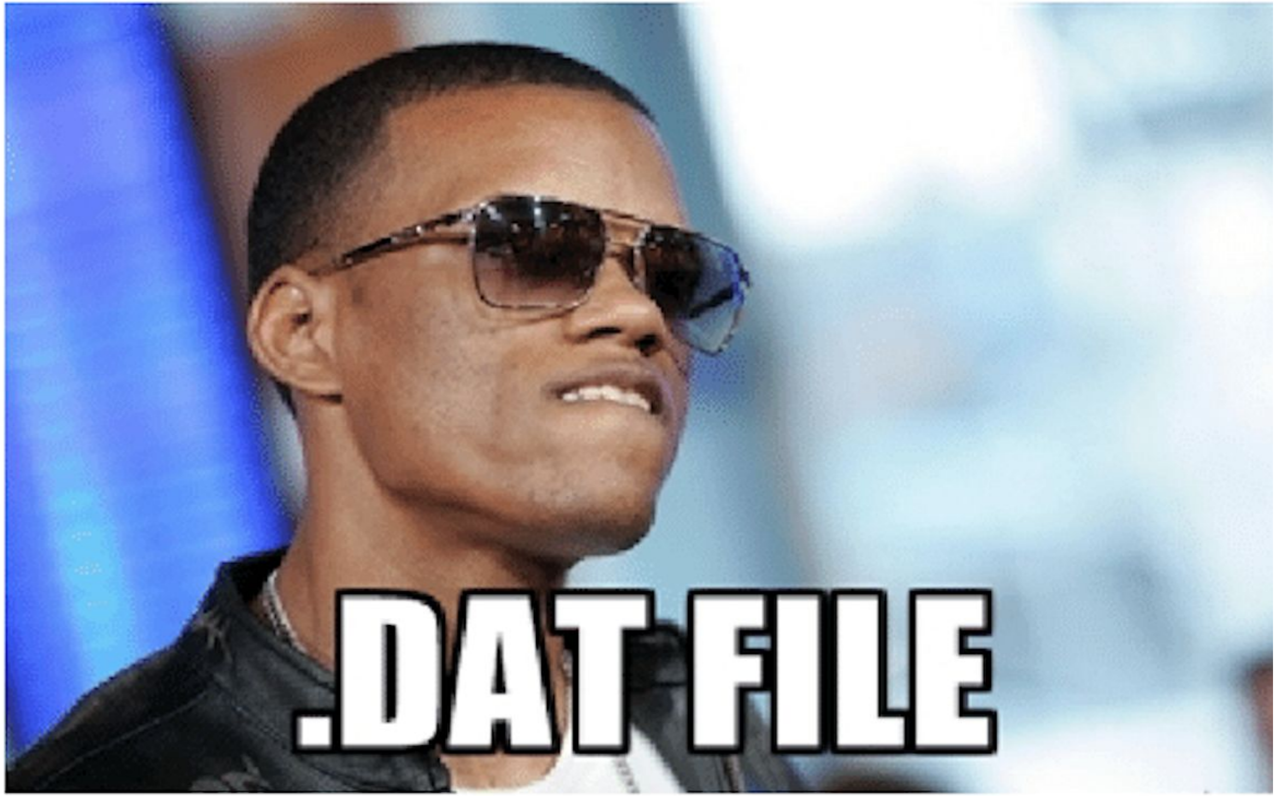
Signals (Contd.)

- Sending the same signal to the parent in all the calls to kill() may print 1 since there would be no queuing of signals.

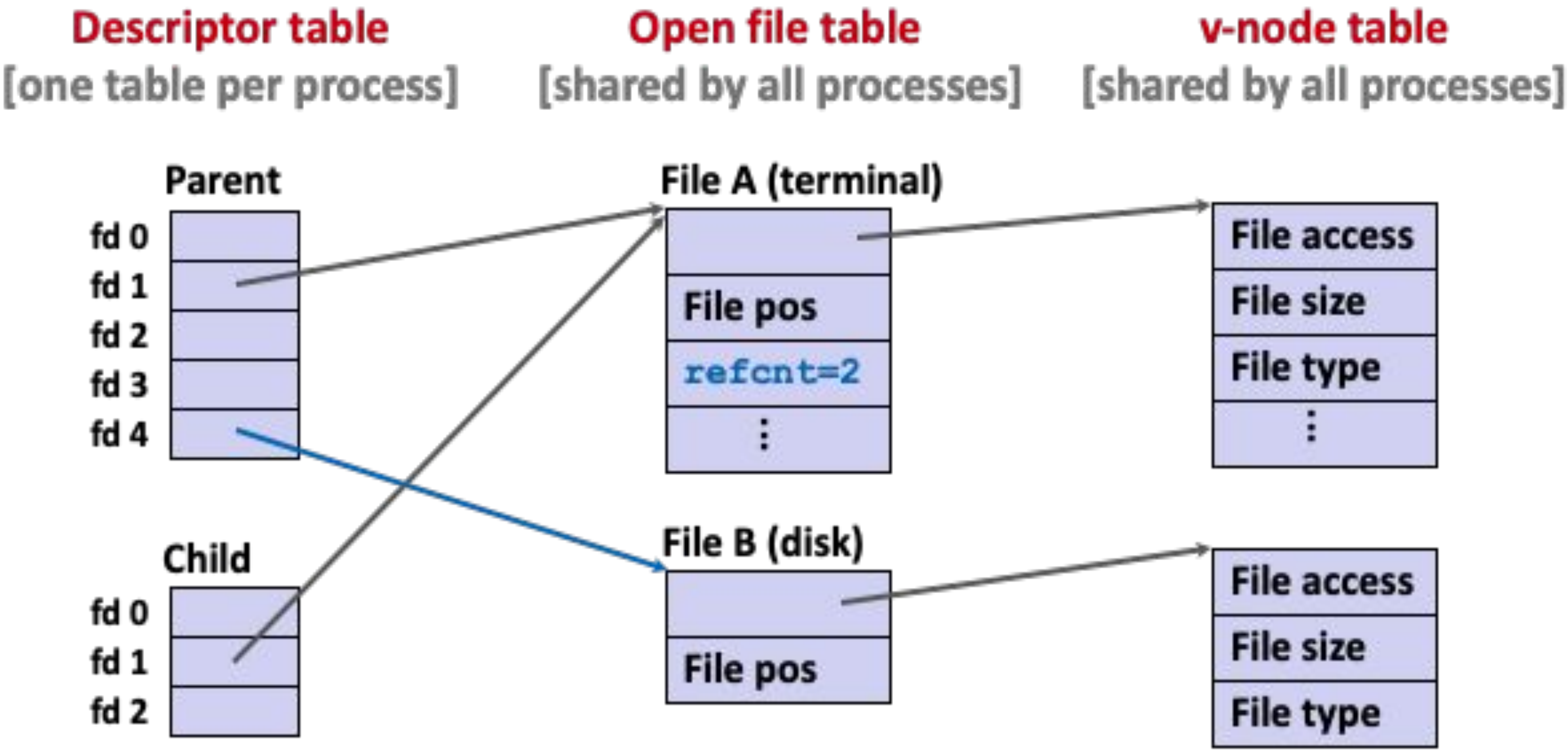
- We can guarantee printing 2 if we send precisely one SIGUSR1 and one SIGUSR2.

- We can print 1-4 depending on the manner in which signals are sent and received.

File IO



How the Unix Kernel Represents Open Files



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?

Just ignore the possible outcomes due to interleaving … try two simple cases :

1. First child executes to the end
2. First parent executes to the end.

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 = 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));
     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 = 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

```
    .
    .
    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.

Malloc

Malloc

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

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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			

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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)						

final-f01.pdf: Problem 10 Part A

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	64a		
free(b)						

final-f01.pdf: Problem 10 Part A

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	64a		
free(b)	80f [0]		32a	64a		

final-f01.pdf: Problem 10 Part A

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	64a		
free(b)	80f [0]		32a	64a		

Malloc - Best fit

- 16 byte align
- coalesced
- footerless
- 32 min size
- fragmentation?
  - internal
  - (32-16) + (64-48) = 32
  - 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	64a		
free(b)	80f [0]		32a	64a		

Malloc - Best fit

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

- fragmentation?
  - internal
  - (32-16) + (64-48) = 32
  - external
  - 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	64a		
free(b)	80f [0]		32a	64a		