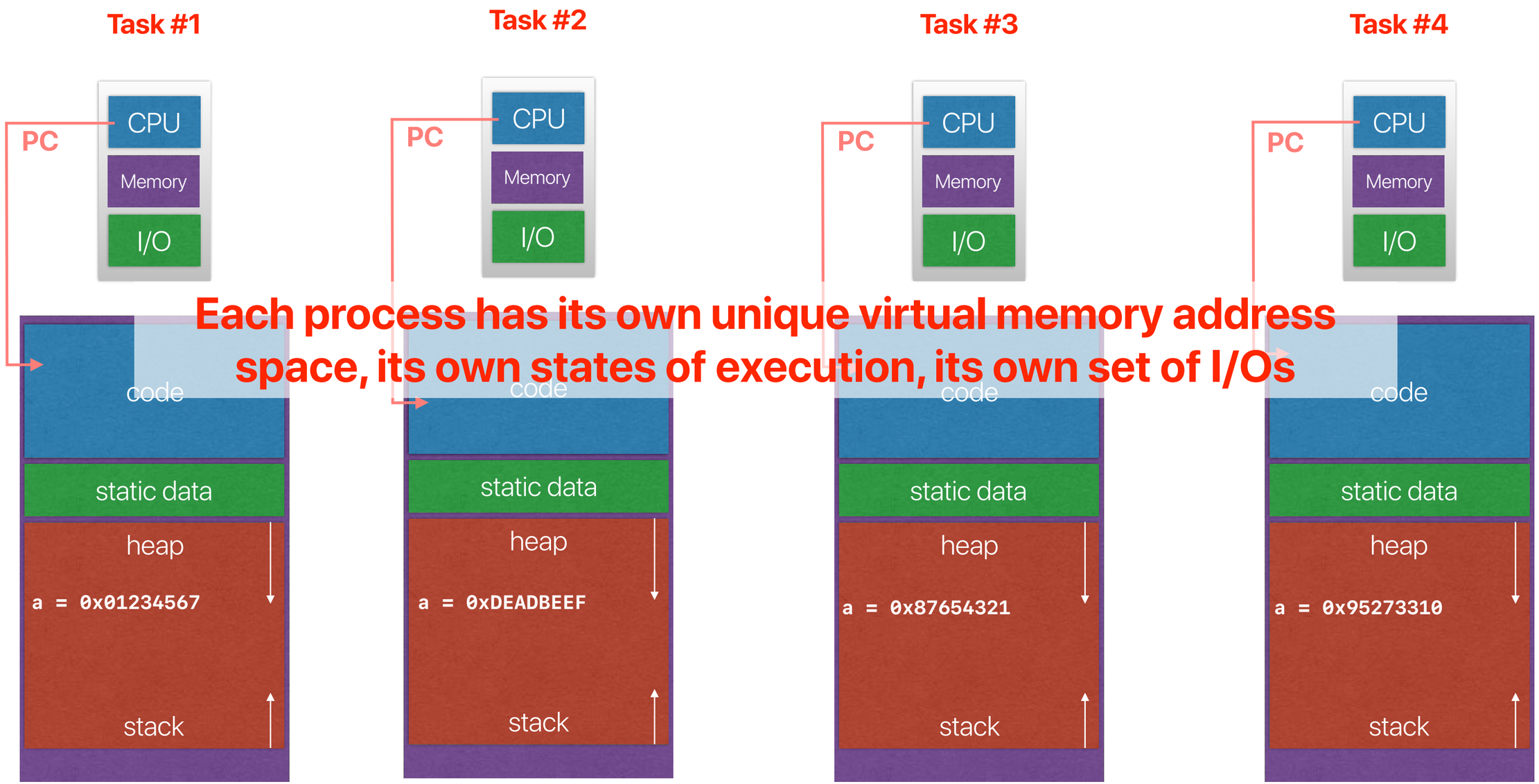
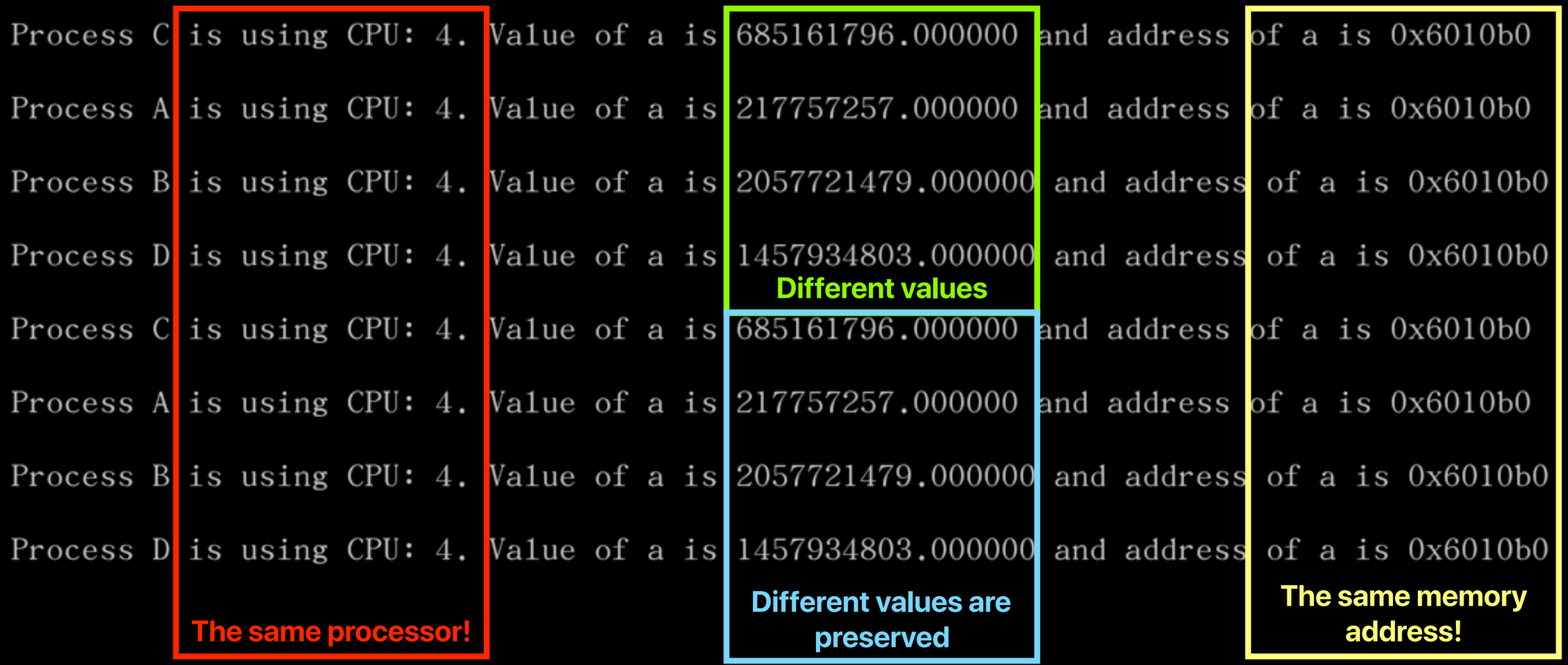
Experiencing processes and threads

Hung-Wei Tseng

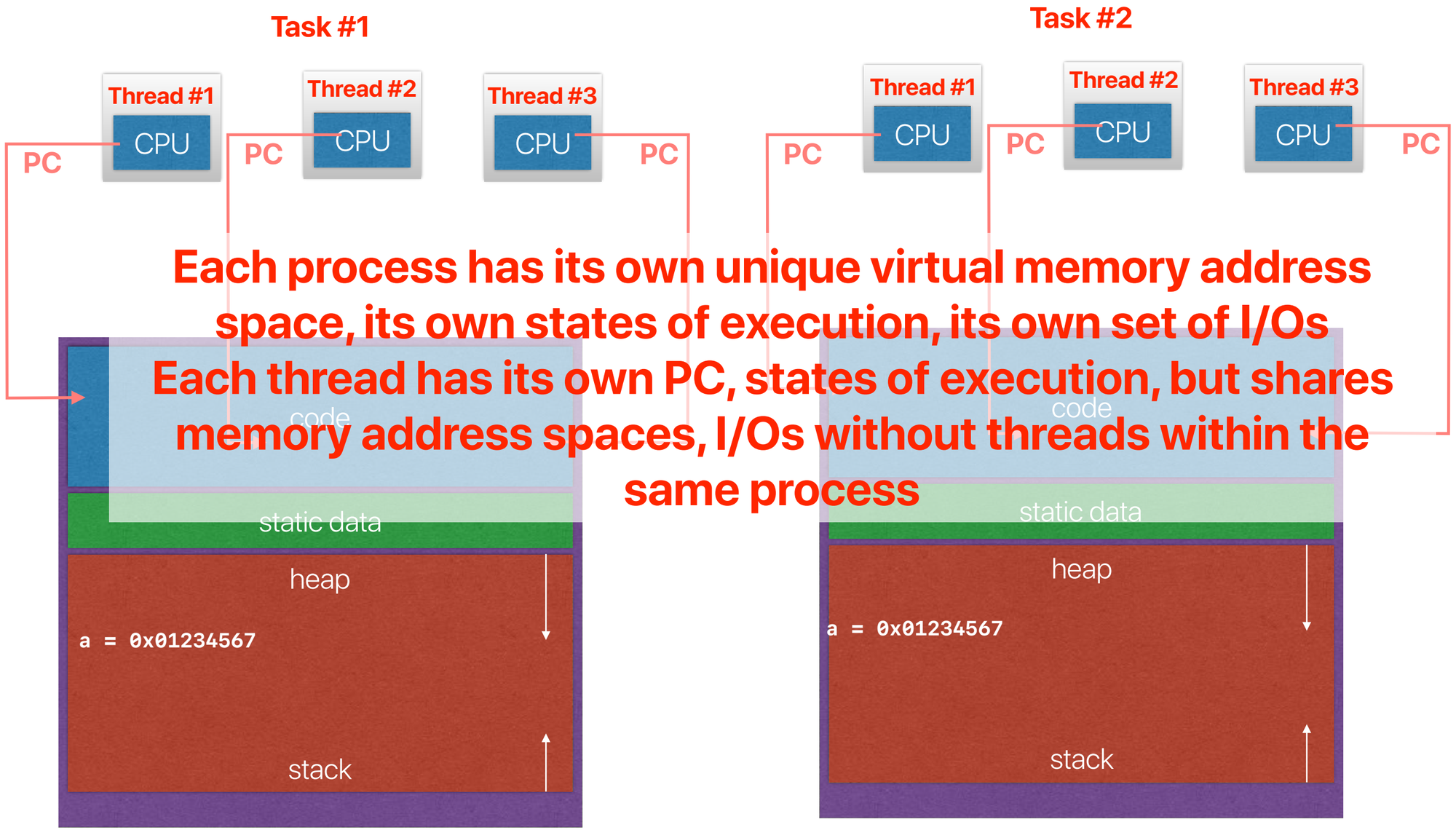
Recap: Tasks/processes



Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os


Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
Process C is using CPU: 4. Value of a is 685161796.000000 and address of a is 0x6010b0
Process A is using CPU: 4. Value of a is 217757257.000000 and address of a is 0x6010b0
Process B is using CPU: 4. Value of a is 2057721479.000000 and address of a is 0x6010b0
Process D is using CPU: 4. Value of a is 1457934803.000000 and address of a is 0x6010b0
Different values
The same processor!
Different values are preserved
The same memory address!

Recap: Threads



Each process has its own unique virtual memory address space, its own states of execution, its own set of I/Os

Each thread has its own PC, states of execution, but shares memory address spaces, I/Os without threads within the same process

Recap: Parallelizing an application

- Threads + shared memory in a process
- Processes + IPC/files/sockets
- Processes + special shared memory module/library (e.g., boost)

Recap: process API — `fork()`

- `pid_t fork();`
- `fork` used to create processes (UNIX)
- What does `fork()` do?
  - Creates a **new** address space (for child)
  - **Copies** parent's address space to child's
  - Points kernel resources to the parent's resources (e.g. open files)
  - Inserts child process into ready queue
- `fork()` returns twice
  - Returns the child's PID to the parent
  - Returns "0" to the child

Outline

- Process API
- Thread programming and synchronization

The interface of managing processes (cont.)

Recap: What will happen?

- What happens if we execute the following code?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf ("My pid is %d\n", getpid());
        }
        printf ("Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent's PID is 2;
child's PID is 7.

	# of times "my pid" is printed	my pid values printed	# of times "child pid" is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

`exit()`

- `void exit(int status)`
- `exit` frees resources and terminates the process
  - Runs an functions registered with atexit
  - Flush and close all open files/streams
  - Releases allocated memory.
  - Remove process from kernel data structures (e.g. queues)
- `status` is passed to parent process
  - By convention, 0 indicates “normal exit”

If we add an exit …

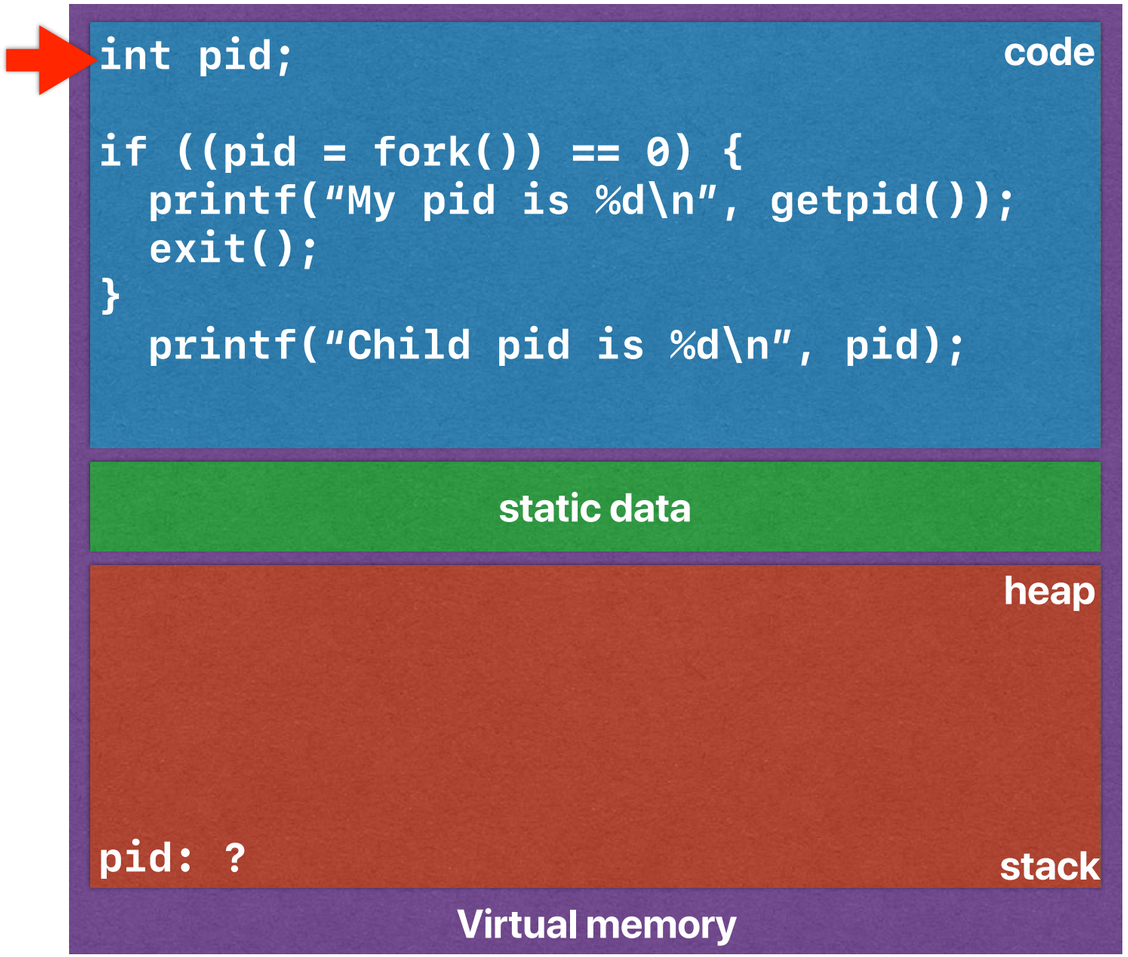
- What happens if we add an exit?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf ("My pid is %d\n", getpid());
            exit(0);
        }
        printf ("Child pid is %d\n", pid);
        return 0;
}
```

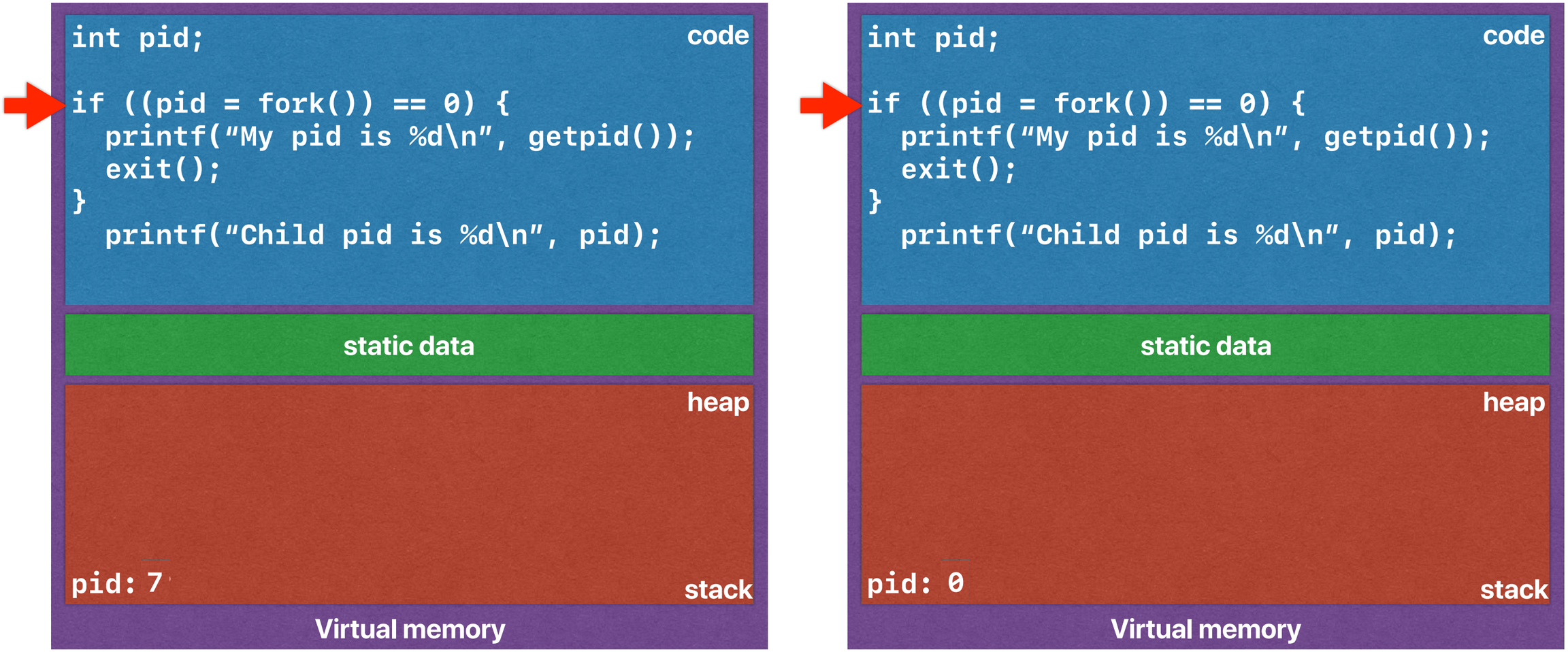
Assume
the parent's PID is 2;
child's PID is 7.

	# of times "my pid" is printed	my pid values printed	# of times "child pid" is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

fork() and exit()

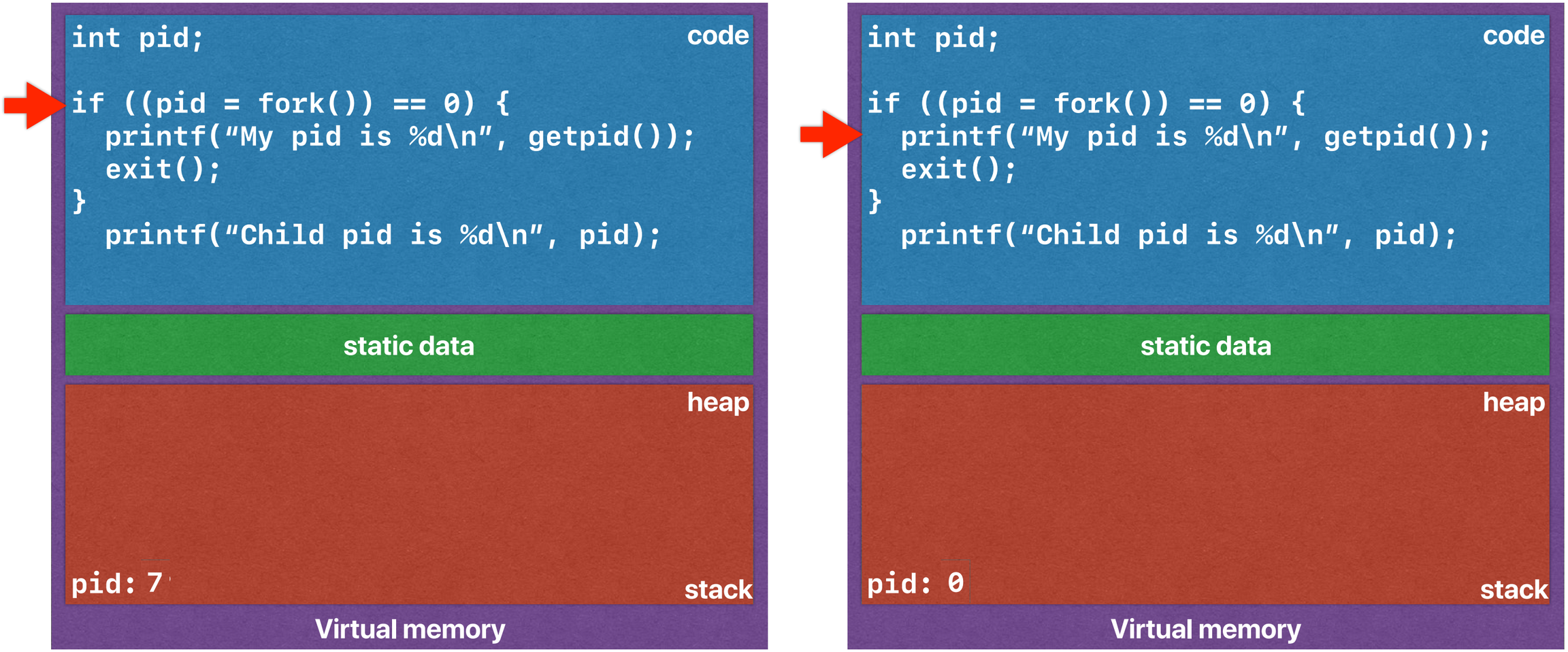


fork() and exit()



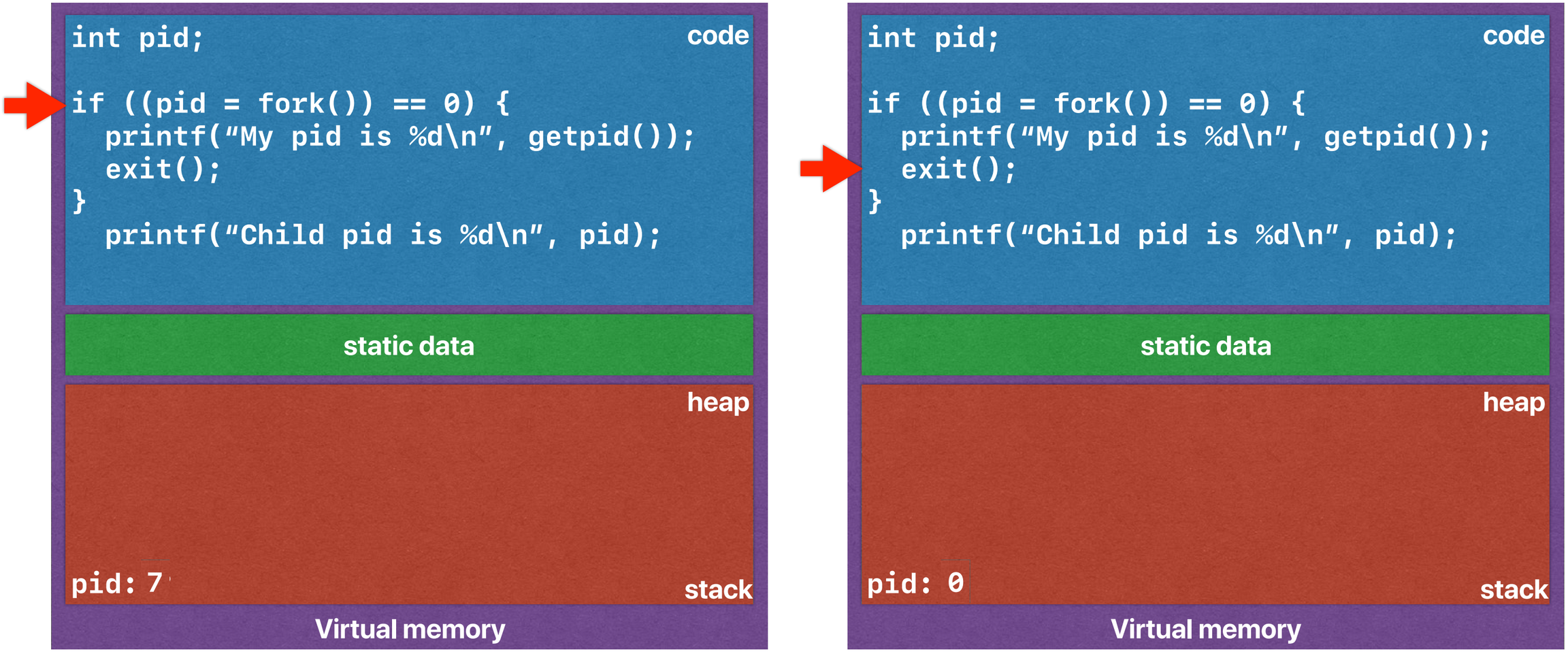
fork() and exit()

Output:
My pid is 7



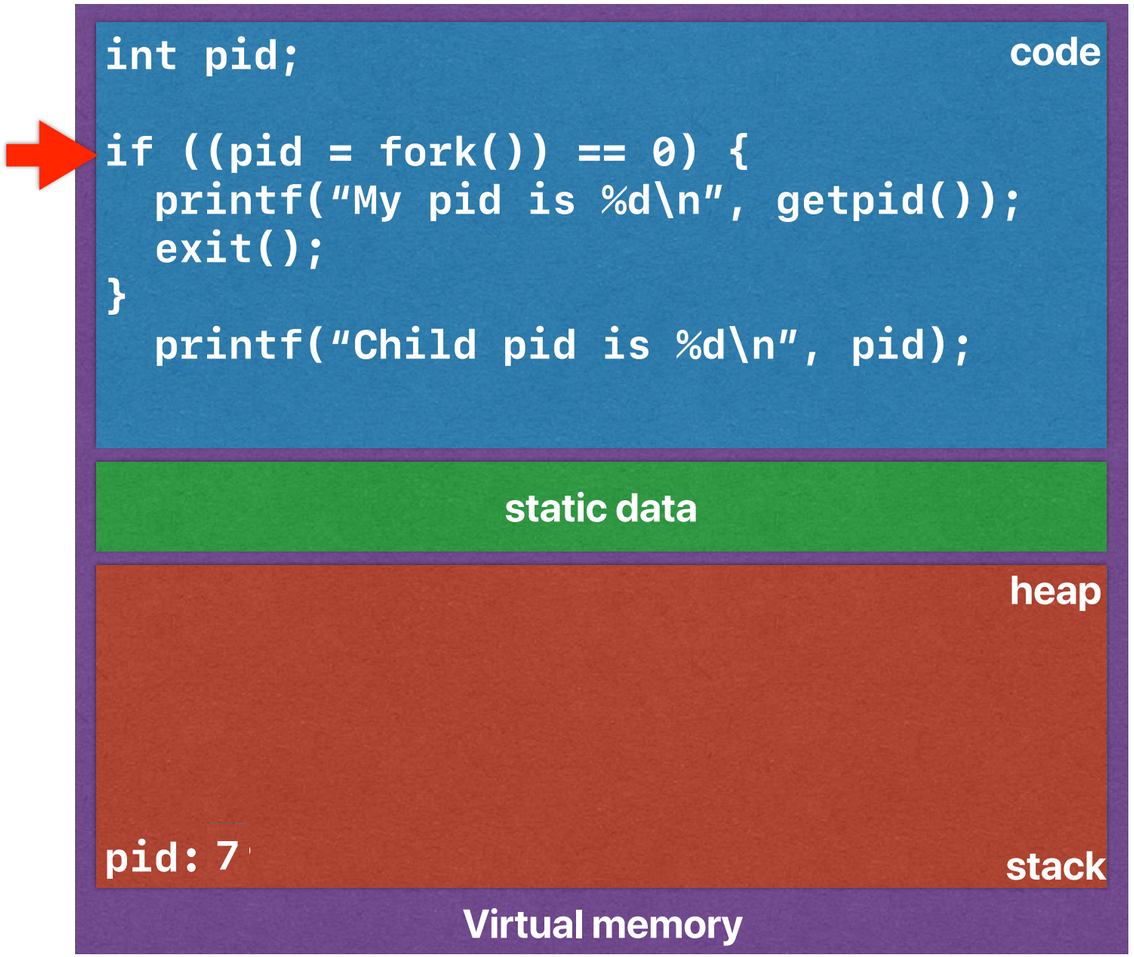
fork() and exit()

Output:
My pid is 7



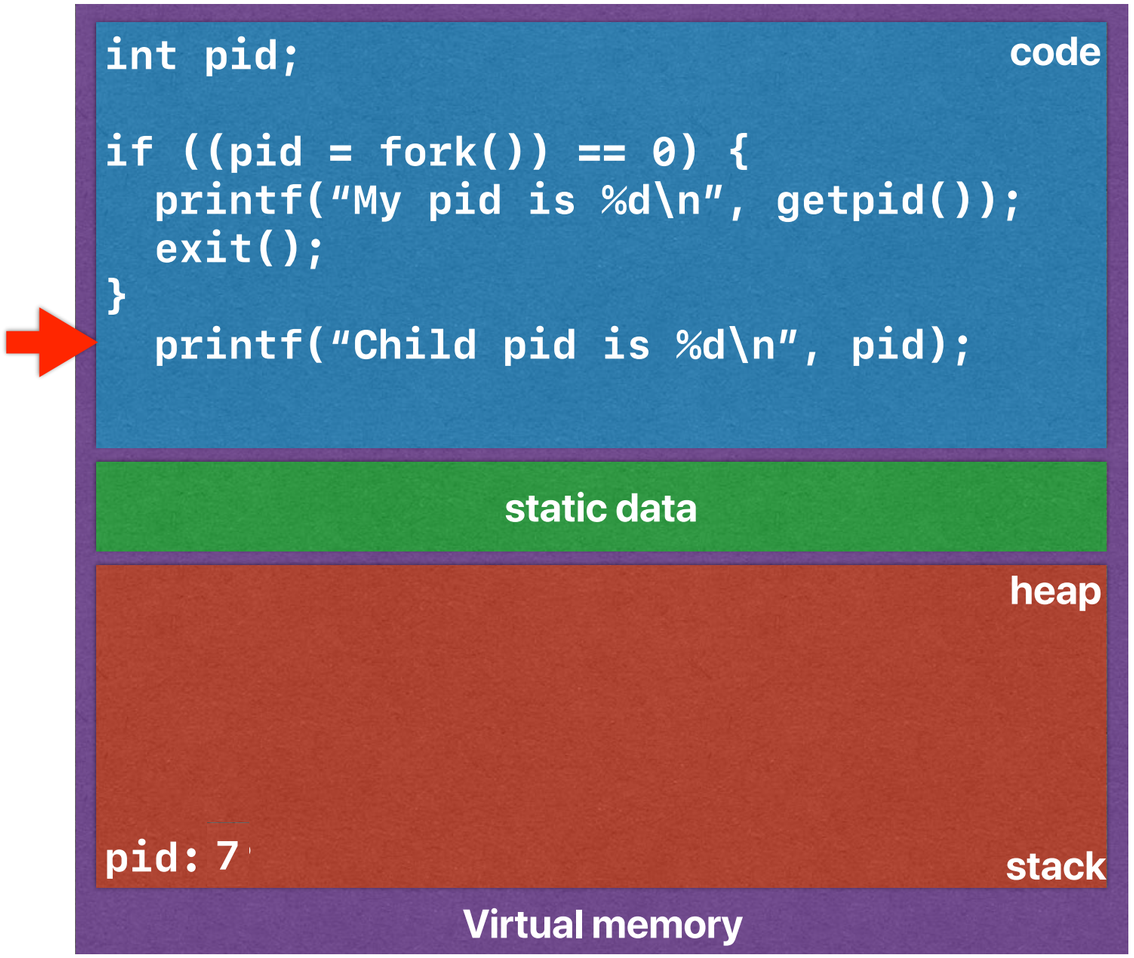
fork() and exit()

Output:
My pid is 7



fork() and exit()

Output:
My pid is 7
Child pid is 7



If we add an exit …

- What happens if we add an exit?

```
int main() {
    int pid;
        if ((pid = fork()) == 0) {
            printf ("My pid is %d\n", getpid());
            exit(0);
        }
        printf ("Child pid is %d\n", pid);
        return 0;
}
```

Assume
the parent's PID is 2;
child's PID is 7.

	# of times "my pid" is printed	my pid values printed	# of times "child pid" is printed	child pid values printed
A	1	7	2	7, 0
B	1	2	2	7, 0
C	2	7, 2	1	7
D	1	0	2	7, 2
E	1	7	1	7

More forks

- Consider the following code

```
fork();
printf("moo\n");
fork();
printf("oink\n");
fork();
printf("baa\n");
```

2x

4x

8x

How many animal noises will be printed?

A. 3

B. 6

C. 8

D. 14

E. 24

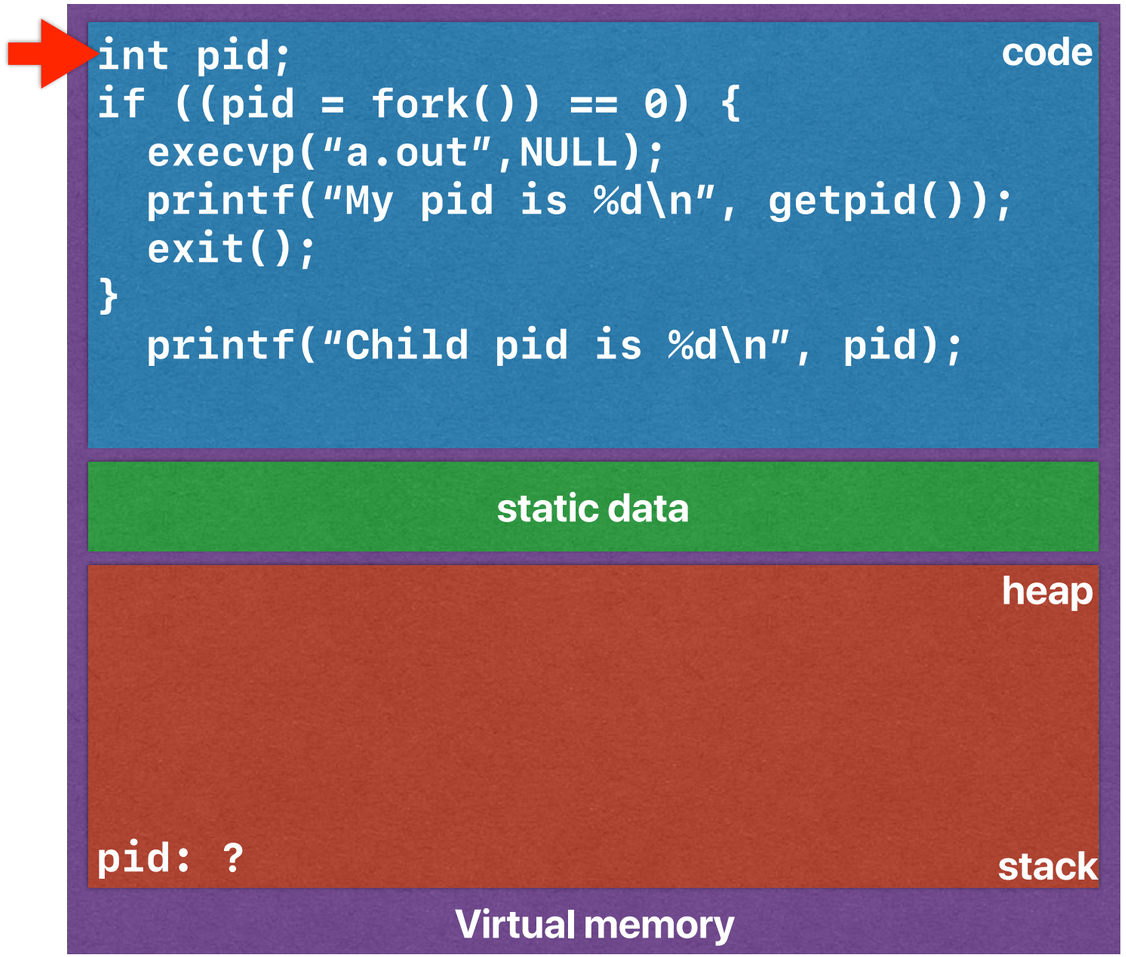
Starting a new program with `execvp()`

- `int execvp(char *prog, char *argv[])`
- `fork` does not start a new program, just duplicates the current program
- What `execvp` does:
  - Stops the current process
  - Overwrites process' address space for the new program
  - Initializes hardware context and args for the new program
  - Inserts the process into the ready queue
- `execvp` does not create a new process

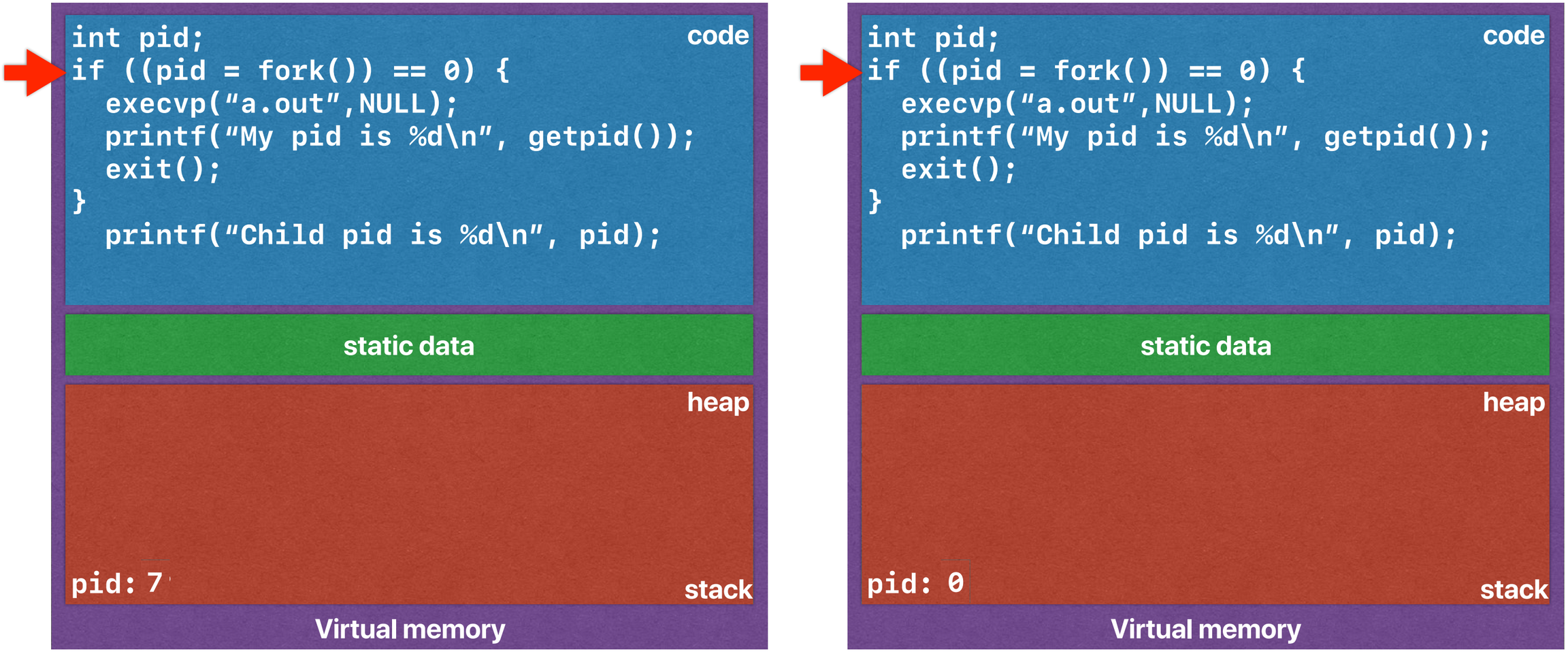
Why separate `fork()` and `exec()`

- Windows only has `exec`
- Flexibility
- Allows redirection & pipe
  - The shell `forks` a new process whenever user invoke a program
  - After `fork`, the shell can setup any appropriate environment variable to before `exec`
  - The shell can easily redirect the output in shell: a.out > file

exec()

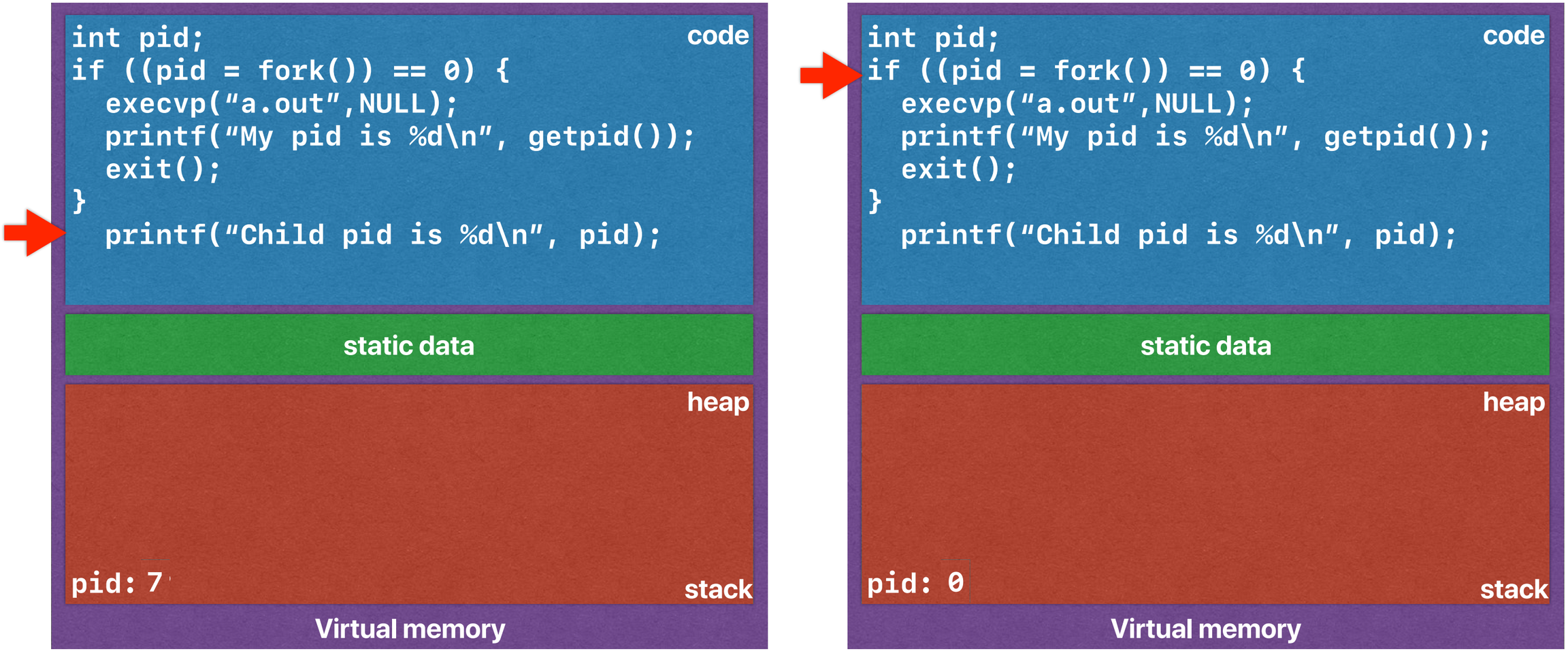

int pid;
if ((pid = fork()) == 0) {
execvp("a.out",NULL);
printf("My pid is %d\n", getpid());
exit();
}
printf("Child pid is %d\n", pid);
code
static data
heap
pid: ?
stack
Virtual memory

fork() and exit()



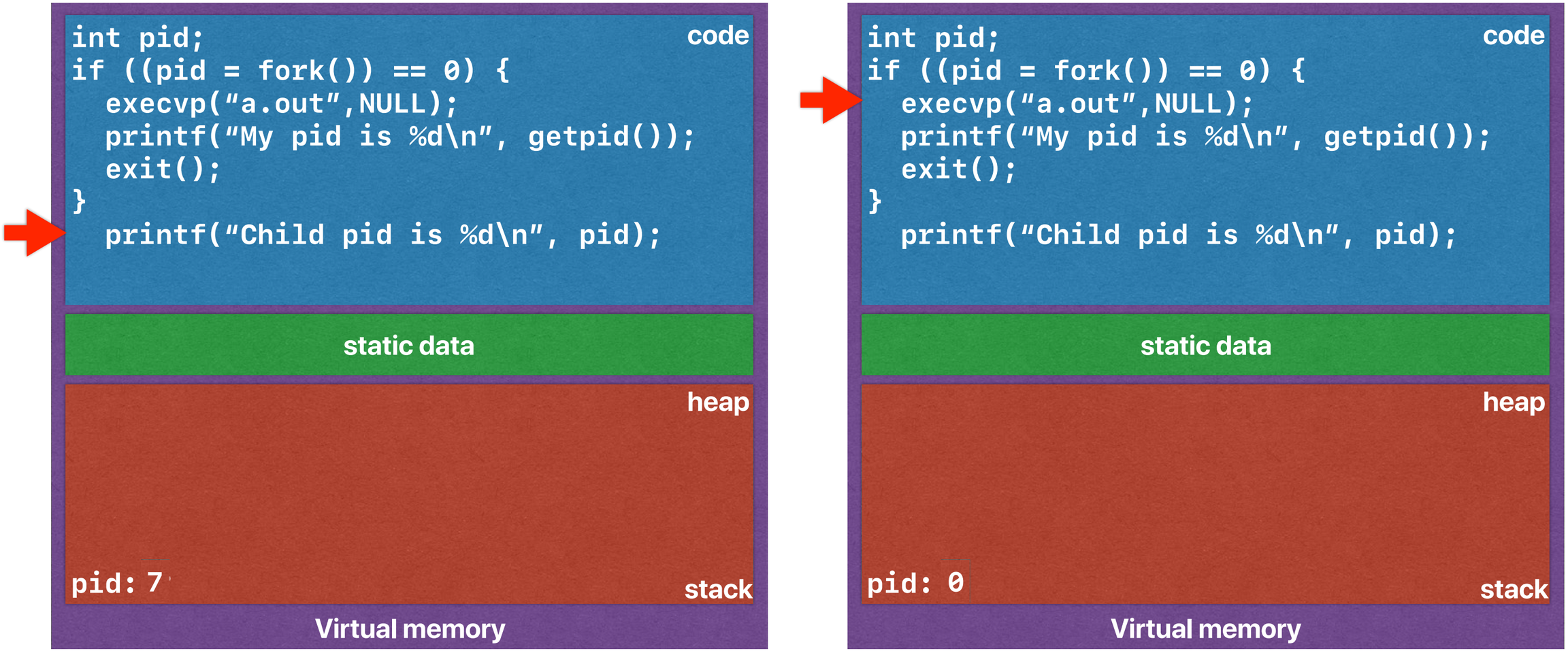
fork() and exit()

Output:
Child pid is 7



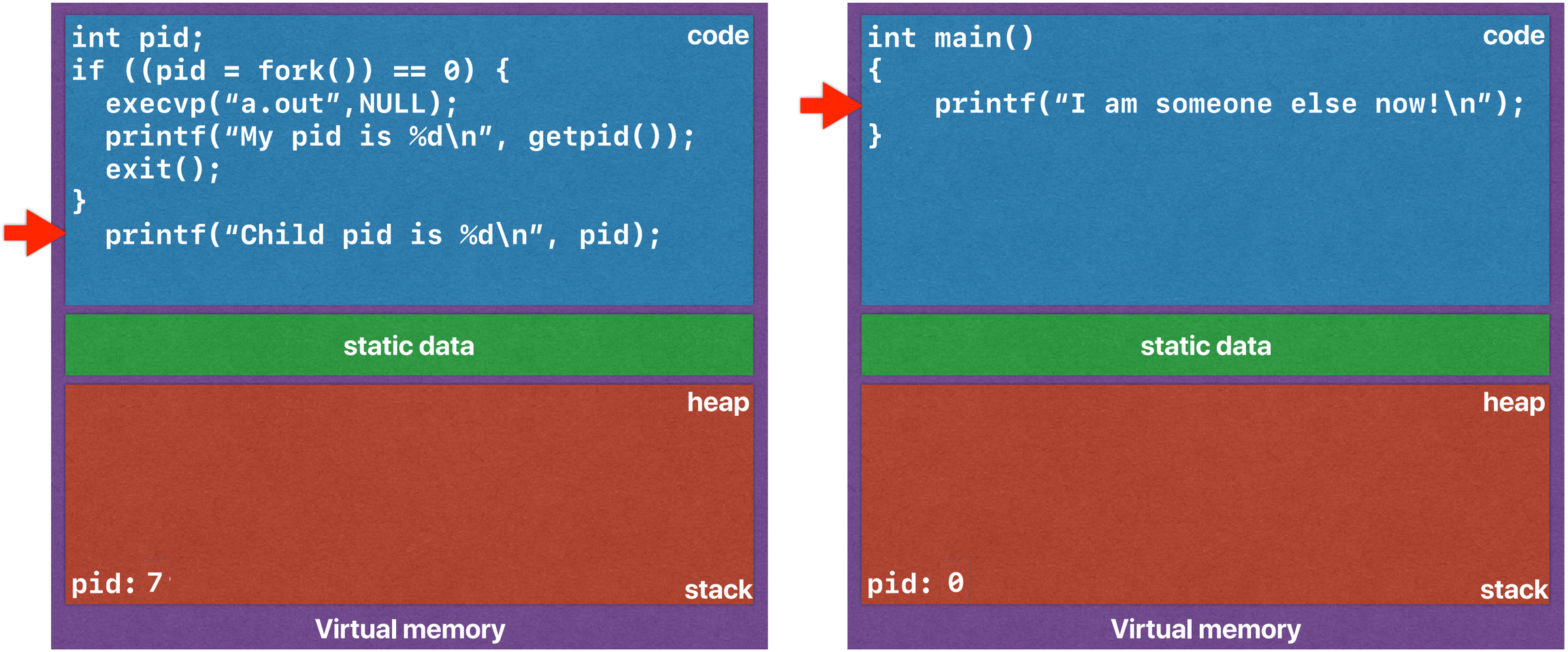
fork() and exit()

Output:
Child pid is 7



fork() and exit()

Output:
Child pid is 7
I am someone else now!



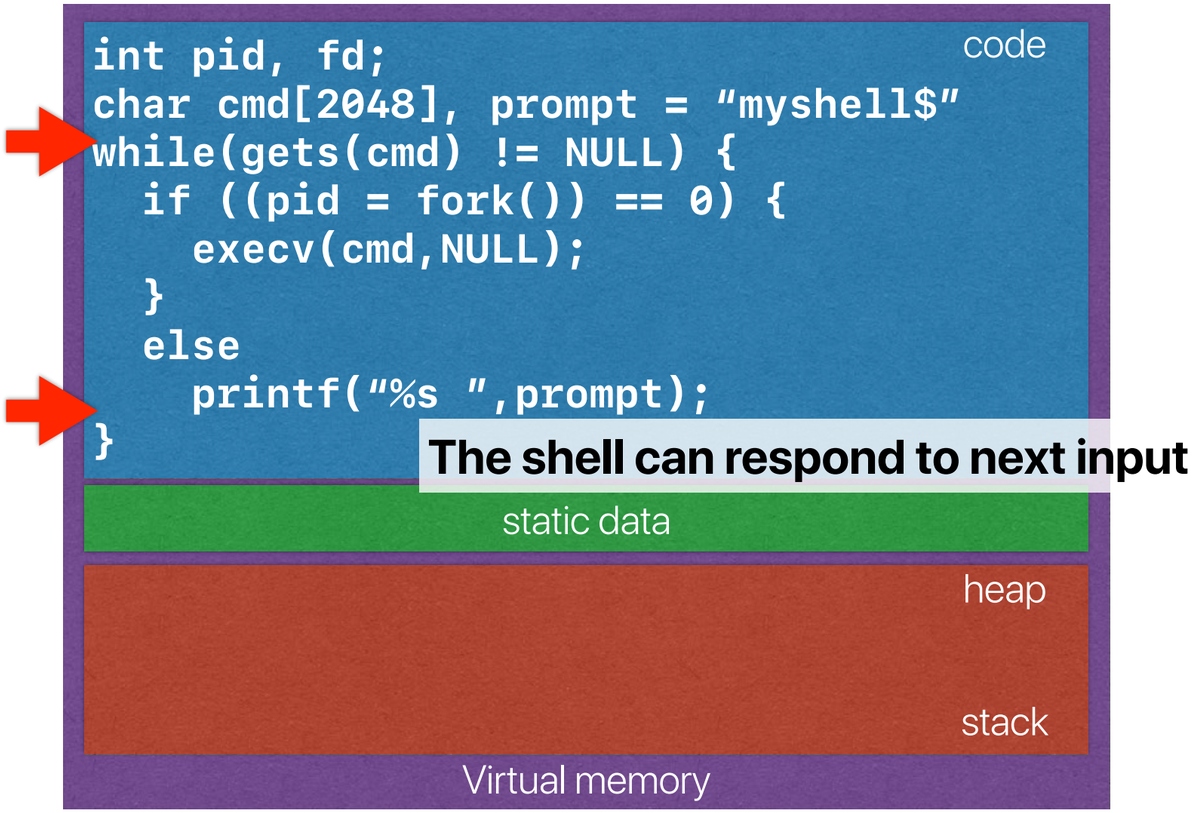
Let’s write our own shells

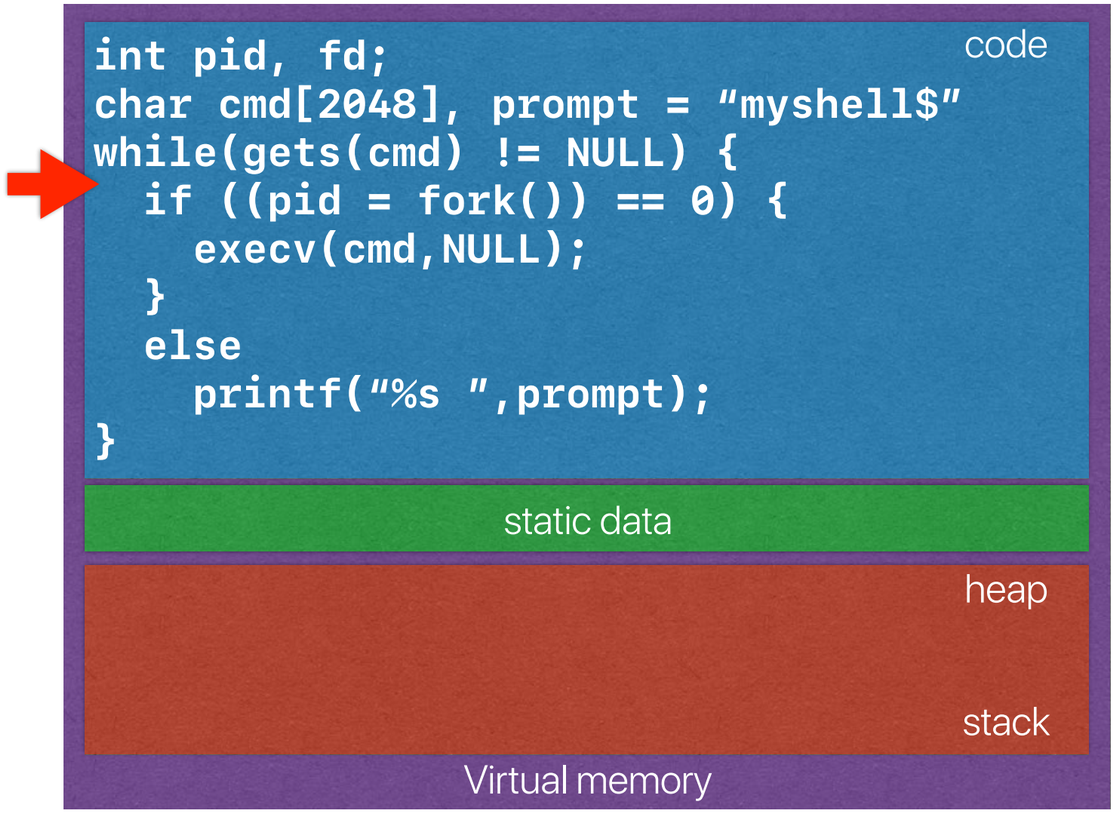
How to implement a shell

- Say, we want to do ./a
- fork
- exec("./a", NULL)

How to implement redirection in shell

- Say, we want to do ./a
- fork
- exec("./a", NULL)





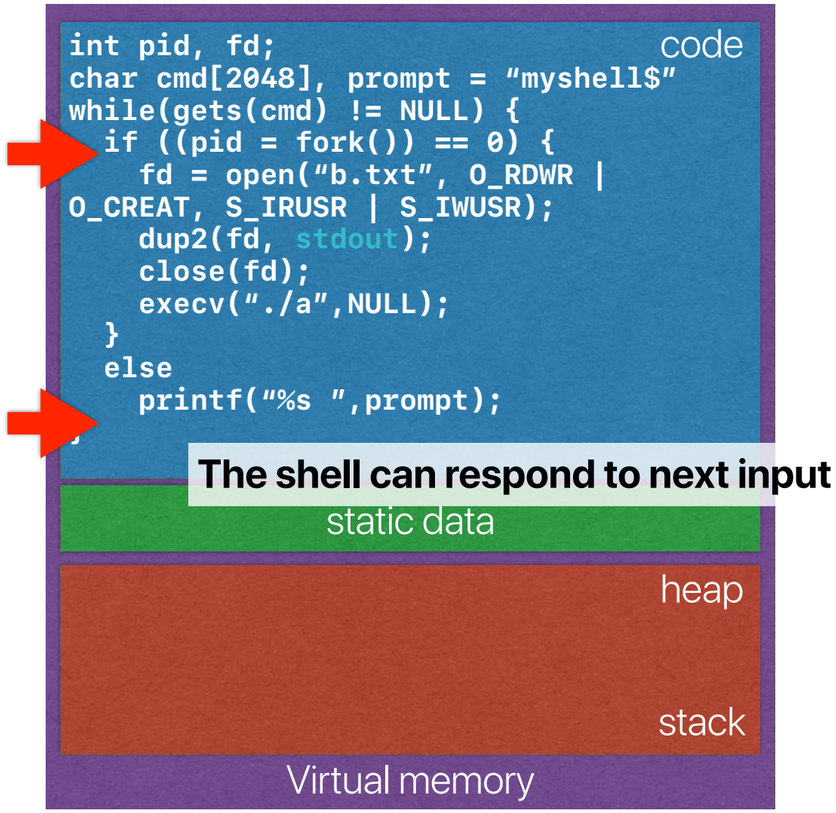
How to implement redirection in shell

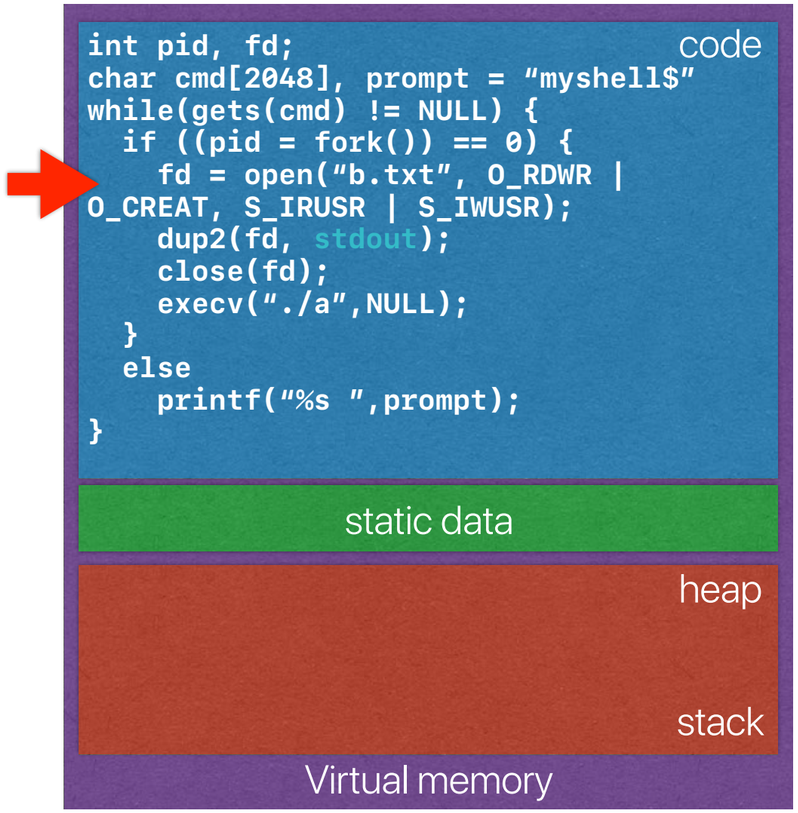
- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor
- The forked code assigns b.txt to stdin/stdout
- The forked code closes b.txt
- exec("./a", NULL)

How to implement redirection in shell

- Say, we want to do ./a > b.txt
- fork
- The forked code opens b.txt
- The forked code dup the file descriptor to stdin/stdout
- The forked code closes b.txt
- exec("./a", NULL)

Homework for you:
Think about the case when
your `fork` is equivalent to `fork+exec()`





wait()

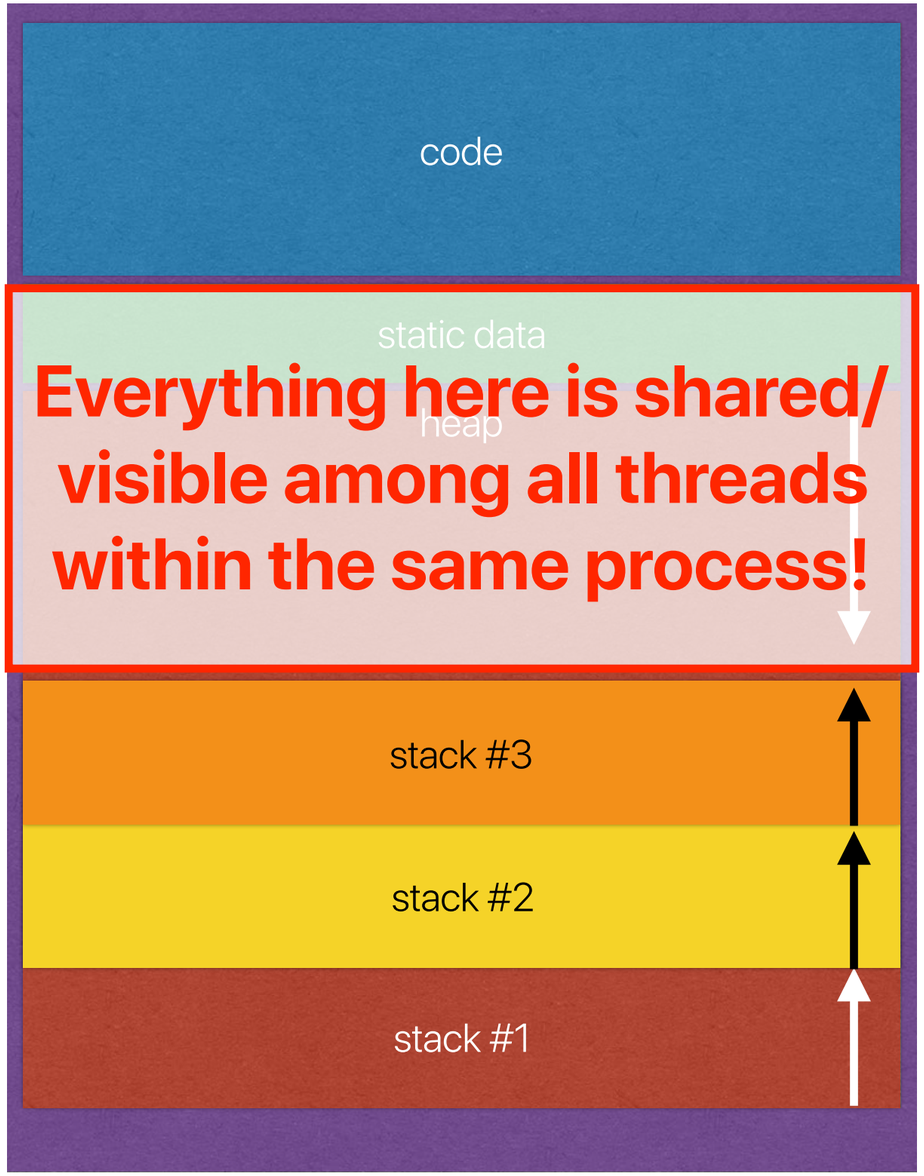
- `pid_t wait(int *stat)`
- `pid_t waitpid(pid_t pid, int *stat, int opts)`
- `wait` / `waitpid` suspends process until a child process ends
  - `wait` resumes when any child ends
  - `waitpid` resumes with child with pid ends
  - `exit` status info 1 is stored in *stat
  - Returns pid of child that ended, or -1 on error
- Unix requires a corresponding `wait` for every `fork`

Zombies, Orphans, and Adoption

- Zombie: process that exits but whose parent doesn't call wait
  - Can't be killed normally
  - Resources freed but pid remains in use
- Orphan: Process whose parent has exited before it has
  - Orphans are **adopted** by init process, which calls wait periodically

Thread programming & synchronization

The virtual memory of multithreaded applications



Joint Banking

withdraw
$20

Bank of UCR

current balance: $40

What is the new balance each would see?

deposit
$10

Bank Account Race Condition

current balance: $50

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

current balance: $40

Thread B

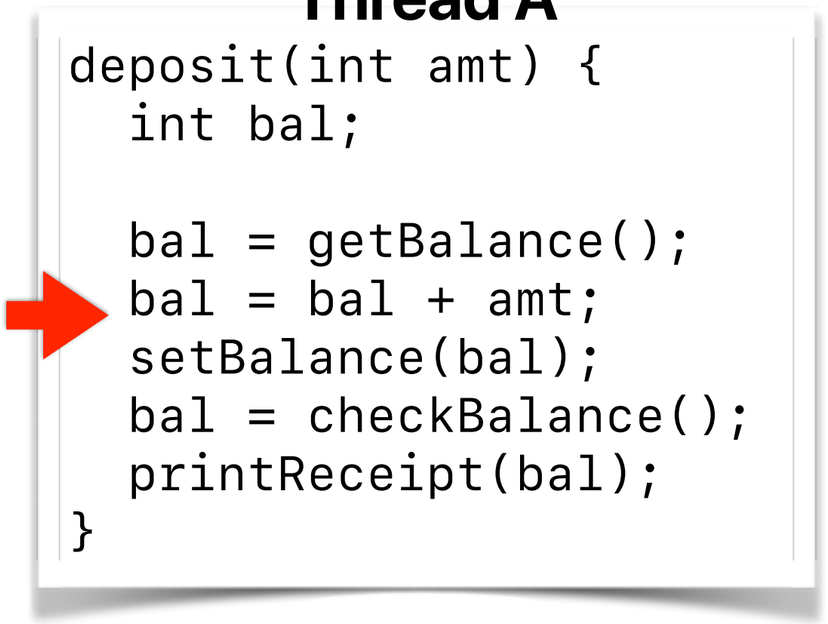
```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Step 1: T_A runs and calls `getBalance` followed by adding `amt` (10) to `bal`

Bank Account Race Condition

current balance: $50

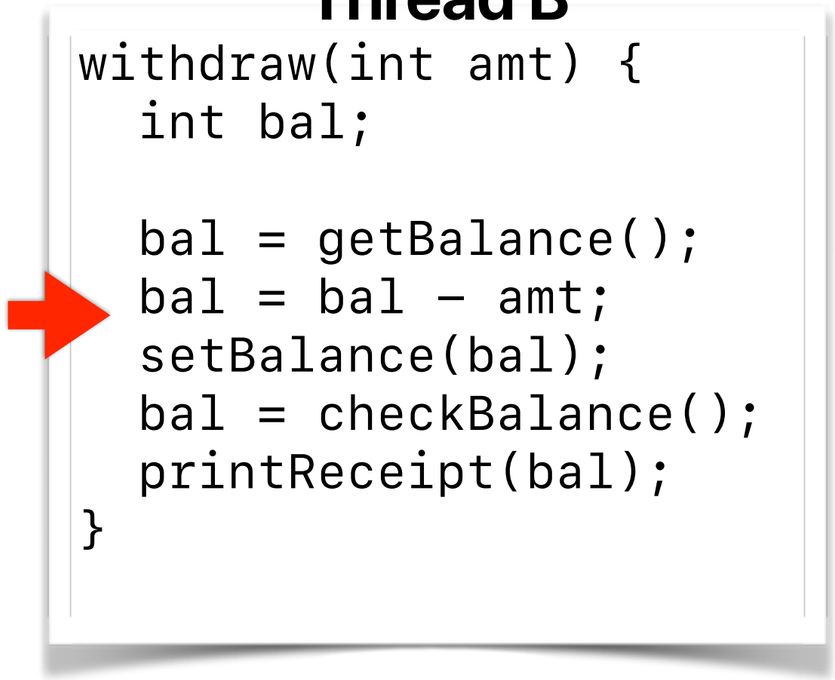
Thread A



```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

current balance: $20

Thread B



```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Step 2: Switch to T_B which calls `getBalance`, followed by subtracting `amt` (20) from `bal`

Bank Account Race Condition

current balance: $50

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

current balance: $20

Thread B

```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Step 3: Switch back to T_A which calls `setBalance`

Bank Account Race Condition

current balance: $50

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

current balance: $20

Thread B

```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Step 4: Switch back to and finish T_B by calling `setBalance`, followed by `printReceipt`

Bank Account Race Condition

current balance: $20

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

current balance: $20

Thread B

```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Step 5: Finish T_A by calling `checkBalance`, followed by `printReceipt`

Honey, we need to chat

Joint Banking

- If the shared variable, balance, initially has the value of 40, what value(s) might it hold after threads A and B finish after we call `deposit(10)` and `withdraw(20)`?

A. 30
B. 20 or 30
C. 20, 30, or 50
D. 10, 20, or 30

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Thread B

```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Processors/threads are not-deterministic

- Processor/compiler may reorder your memory operations/ instructions
- Each processor core may not run at the same speed (cache misses, branch mis-prediction, I/O, voltage scaling and etc..)
- Threads may not be executed/scheduled right after it's spawned

Synchronization

- Concurrency leads to multiple active processes/threads that share one or more resources
- Synchronization involves the orderly sharing of resources
- All threads must be on the same page
- Need to avoid race conditions

Critical sections

- Protect some pieces of code that access shared resources (memory, device, etc.)
- For safety, critical sections should:
  - Enforce mutual exclusion (i.e. only one thread at a time)
  - Execute atomically (all-or-nothing) before allowing another thread

Identifying Critical Sections

- Which is the smallest code region in Thread A that we can make as a critical section to guarantee the outcome as 30?

Thread A

```
int deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();

  printReceipt(bal);
  return bal;
}
```

A B C D E

Thread B

```
int withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal - amt;
  setBalance(bal);
  bal = checkBalance();

  printReceipt(bal);
  return bal;
}
```

Critical sections

Thread A

```
deposit(int amt) {
  int bal;

  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Thread B

```
withdraw(int amt) {
  int bal;

  bal = getBalance();
  bal = bal – amt;
  setBalance(bal);
  bal = checkBalance();
  printReceipt(bal);
}
```

Critical section

- Entry section acts as barrier, only allowing a single thread in at a time
- Exit section should remove barrier for other threads' entry

```
deposit(int amt) {
  int bal;
  entry section
  bal = getBalance();
  bal = bal + amt;
  setBalance(bal);
  bal = checkBalance();
  exit section
  printReceipt(bal);
}
```

Solving the "Critical Section Problem"

1. Mutual exclusion — at most one process/thread in its critical section
2. Progress/Fairness — a thread outside of its critical section cannot block another thread from entering its critical section
3. Progress/Fairness — a thread cannot be postponed indefinitely from entering its critical section
4. Accommodate nondeterminism — the solution should work regardless the speed of executing threads and the number of processors

How to implement lock/unlock

The baseline

```c
int max;
volatile int balance = 0; // shared global varia
```

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first <loopcount>\n");
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", \ balance,
(unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
     balance, max*2);
    return 0;
}
```

```c
void * mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per
thread)
    printf("%s: begin [addr of i:
%p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        balance = balance + 1; //
shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

Use pthread_lock

```
int max;
volatile int balance = 0; // shared global variable
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first <loopcount>\n");
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      Pthread_mutex_lock(&lock);
      balance++;
      Pthread_mutex_unlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

Use pthread_lock (coarser grain)

```
int max;
volatile int balance = 0; // shared global variable
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first <loopcount>\n");
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
      Pthread_mutex_lock(&lock);
       for (i = 0; i < max; i++) {
          balance++;
       }
      Pthread_mutex_unlock(&lock);
    printf("%s: done\n", letter);
    return NULL;
}
```

Use spin locks

```
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      SpinLock(&lock);
      balance++;
      SpinUnlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
   fprintf(stderr, "usage: main-first <loopcount>\n");
   exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

what if context switch happens here?

```
void SpinLock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}

void SpinUnlock(volatile unsigned int *lock)
    *lock = 0;
}
```

Disable interrupts?

- Which of the following can disabling interrupts guarantee on multicore processors?

What if we have multiple processors?

A. At most one process/thread in its critical section
B. A thread outside of its critical section cannot block another thread from entering its critical section
C. A thread cannot be postponed indefinitely from entering its critical section

What if the thread hold the critical section crashes?

D. The solution should work regardless the speed of executing threads and the number of processors
E. None of the above

What if we have multiple processors?

Use spin locks

```
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
      SpinLock(&lock);
      balance++;
      SpinUnlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
   fprintf(stderr, "usage: main-first <loopcount>\n");
   exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %d] [%x]\n", balance,
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
      balance, max*2);
    return 0;
}
```

what if context switch happens here?

— the lock must be updated atomically

```
void SpinLock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    // spin
    *lock = 1;          // SET (lock)
}

void SpinUnlock(volatile unsigned int *lock)
    *lock = 0;
}
```

Use spin locks

```
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
```

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythrea
    Pthread_create(&p2, NULL, mythrea
    // join waits for the threads to
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %
      balance, max*2);
    return 0;
}
```

```
static inline uint xchg(volatile unsigned int *addr,
unsigned int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}

void SpinLock(volatile unsigned int *lock) {
    // what code should go here?
}

void SpinUnlock(volatile unsigned int *lock) {
    // what code should go here?
}
```

exchange the content in %0 and %1

a prefix to xchgl that locks the whole cache line

Use spin locks

```
int max;
volatile int balance = 0; // shared global variable
volatile unsigned int lock = 0;
```

```
void * mythread(void *arg) {
    char *letter = arg;
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
```

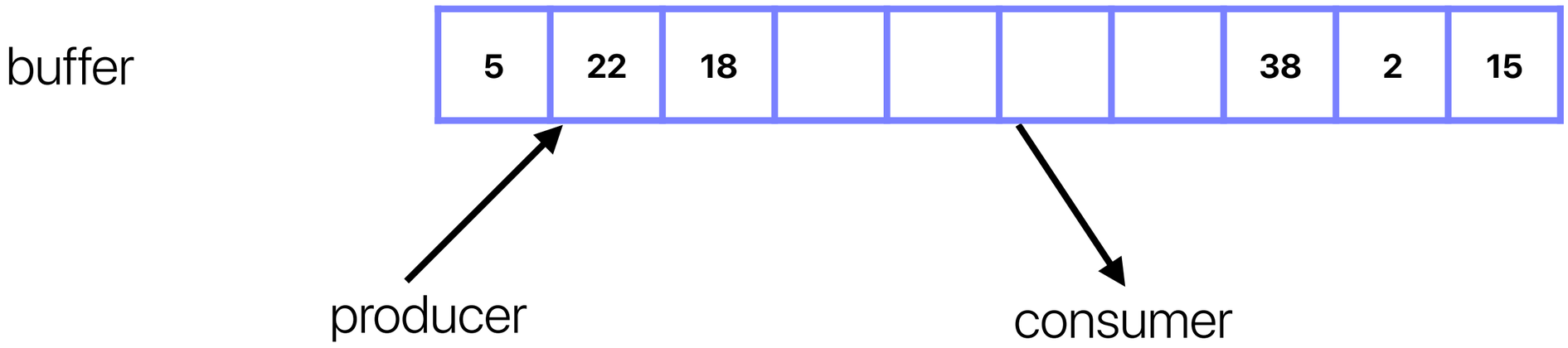
```
int main(int argc, char *argv[])
{
    if (argc != 2) {
  fprintf(stderr, "usage: main-first
  exit(1);
    }
    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [balance = %
      (unsigned int) &balance);
    Pthread_create(&p1, NULL, mythrea
    Pthread_create(&p2, NULL, mythrea
    // join waits for the threads to
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %
      balance, max*2);
    return 0;
}
```

```
static inline uint xchg(volatile unsigned int *addr, unsigned
int newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" : "+m" (*addr),
"=a" (result) : "1" (newval) : "cc");
    return result;
}

void SpinLock(volatile unsigned int *lock) {
    while (xchg(lock, 1) == 1);
}

void SpinUnlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
```

Semaphores

Bounded-Buffer Problem

- Also referred to as "producer-consumer" problem
- Producer places items in shared buffer
- Consumer removes items from shared buffer



Without synchronization, you may write

```
int buffer[BUFF_SIZE]; // shared global
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        int item = buffer[out];
        out = (out + 1) %
BUFF_SIZE;
        // do something w/ item
    }
    return NULL;
}
```

Use locks

```
int buffer[BUFF_SIZE]; // shared global
volatile unsigned int lock = 0;
```

```
int main(int argc, char *argv[]) {
    pthread_t p;
    printf("parent: begin\n");
    // init here
    Pthread_create(&p, NULL, child, NULL);
    int in = 0;
    while(TRUE) {
        int item = …;
        Pthread_mutex_lock(&lock);
        buffer[in] = item;
        in = (in + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
    }
    printf("parent: end\n");
    return 0;
}
```

```
void *child(void *arg) {
    int out = 0;
    printf("child\n");
    while(TRUE) {
        Pthread_mutex_lock(&lock);
        int item = buffer[out];
        out = (out + 1) % BUFF_SIZE;
        Pthread_mutex_unlock(&lock);
        // do something w/ item
    }
    return NULL;
}
```

You cannot produce and consume simultaneously!

Announcement

- Reading quiz due next Tuesday
- Project will be up next Tuesday after class
  - You need to install Ubuntu 16.04.6 in VirtualBox
  - You can have up to 1 classmate working together with you — 2 in a group