Memory Management Part 5

Quiz 1

Unix process A has private-mapped a file into its address space. Our system has one-byte pages and the file consists of four pages. The pages are mapped into locations 100 through 103. The initial values of these pages are all zeroes.

1) A stores a 1 into location 100
2) A forks, creating process B
3) A stores a 1 into location 101
4) B stores a 2 into location 102
5) B forks, creating process C
6) A stores 111 into location 100
7) B stores 222 into location 103
8) C sums the contents of locations 100, 101, and 102, and stores them into location 103

What value did C store into 103?

Answer:

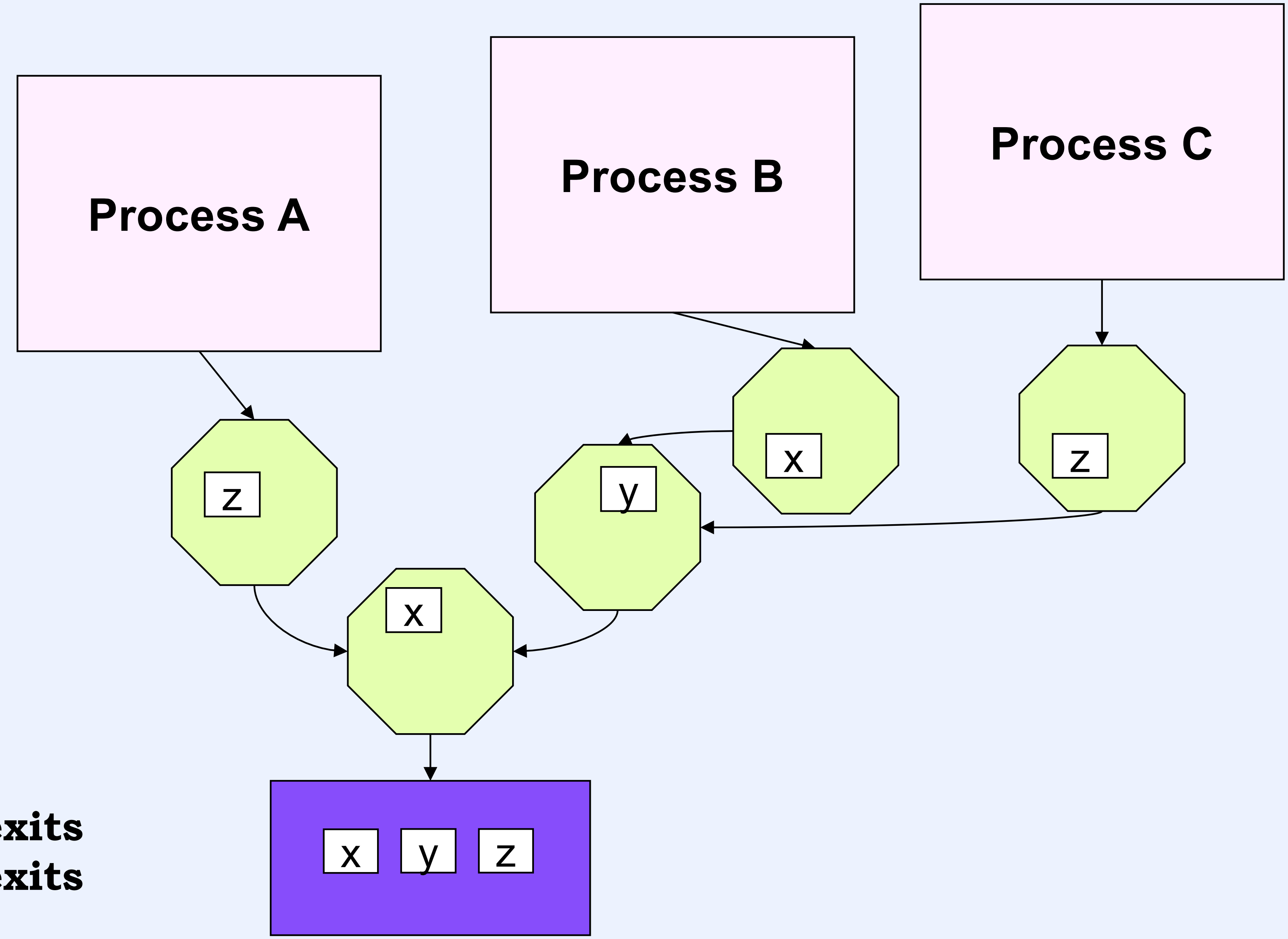
a) 0
b) 3
c) 4
d) 113

Fork Bomb!

```
int main() {
  while (1) {
    if (fork() == 0)
      exit(0);
  }
  return 0;
}
```

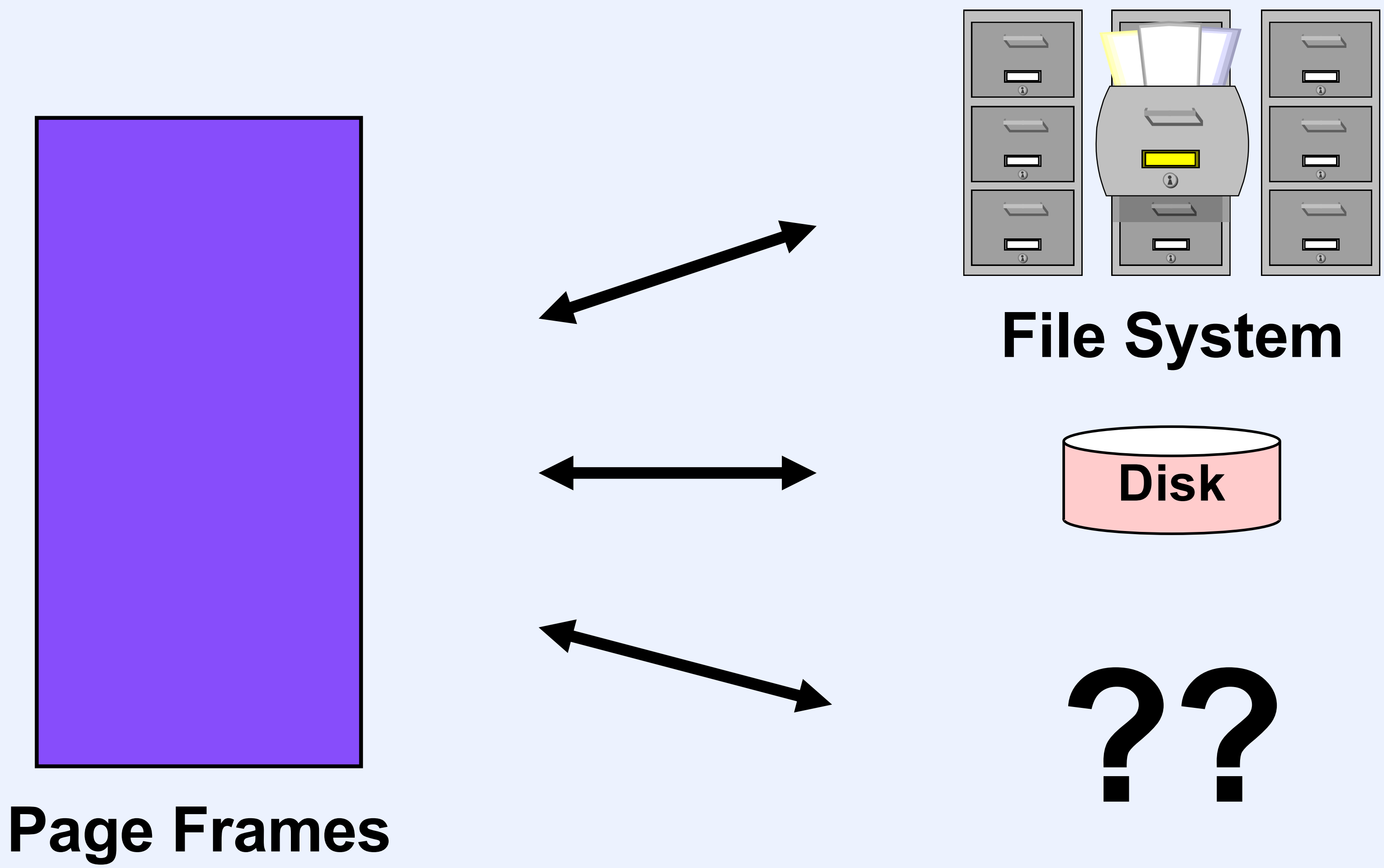
```
int main() {
  while (1) {
    if (fork() > 0)
      exit(0);
  }
  return 0;
}
```

Private Mapping (Continued)



Process B exits
Process A exits

The Backing Store



Backing Up Pages (1)

- Read-only mapping of a file (e.g. text)
  - pages come from the file, but, since they are never modified, they never need to be written back
- Read-write shared mapping of a file (e.g. via *mmap* system call)
  - pages come from the file, modified pages are written back to the file

Backing Up Pages (2)

- **Read-write private mapping of a file (e.g. the data section as well as memory mapped private by the *mmap* system call)**
  - **pages come from the file, but modified pages, associated with shadow objects, must be backed up in swap space**
- **Anonymous memory (e.g. bss, stack, and shared memory)**
  - **pages are created as *zero fill on demand*; they must be backed up in swap space**

Swap Space

- Space management possibilities
  - radical-conservative approach: pre-allocation
    - backing-store space is allocated when virtual memory is allocated
    - page outs always succeed
    - might need to have much more backing store than needed
  - radical-liberal approach: lazy evaluation
    - backing-store space is allocated only when needed
    - page outs could fail because of no space
    - can get by with minimal backing-store space

Space Allocation in Linux

- Total memory = primary + swap space
- System-wide parameter: *overcommit_memory*
  - three possibilities
    - maybe (default)
    - always
    - never
- mmap has MAP_NORESERVE flag
  - don’t worry about over-committing

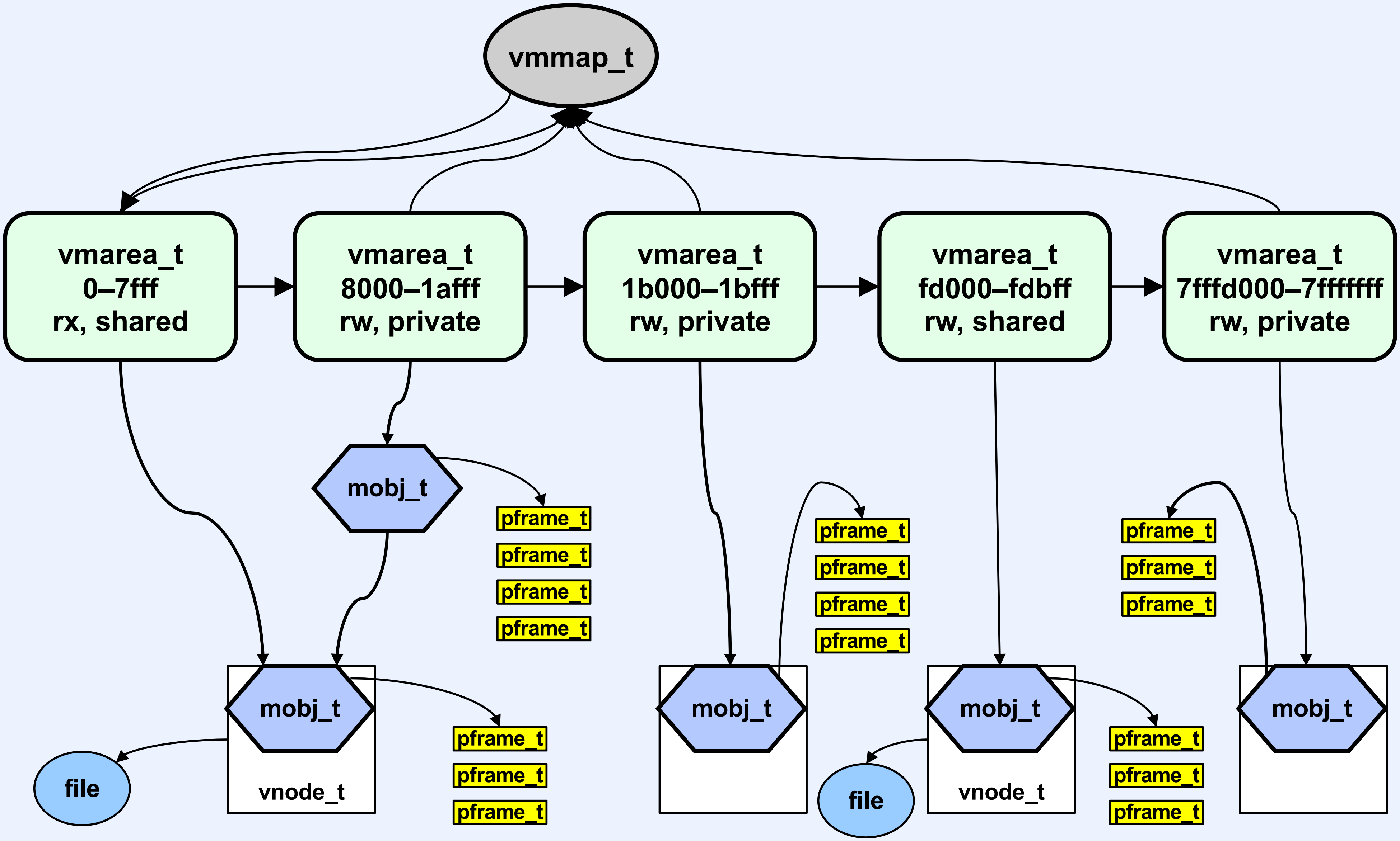
Space Allocation in Windows

- Space reservation
  - allocation of virtual memory
- Space commitment
  - reservation of physical resources
    - paging space + physical memory
- *MapViewOfFile* (sort of like *mmap*)
  - no over-commitment
- Thread creation
  - creator specifies both reservation and commitment for stack pages

Space Allocation in Weenix

- Shadow memory objects
  – no backing store
  – pages exist in primary memory only
- Anonymous memory objects
  – used for virtual memory not mapped to a file
    - BSS, dynamic, stack
  – no backing store
  – pages exist in primary memory only

Weenix Address Space



Quiz 2

A page containing initialized global data is accessed for the first time by the process. It will be cached

a) in the file's mobj

b) in the file system's mobj

c) someplace else

d) not at all

Quiz 3

A page containing uninitialized global data is accessed for the first time by the process. It will be cached

a) in the file's mobj

b) in the file system's mobj

c) someplace else

d) not at all

Quiz 4

A file is created and one byte is written to it at location 2^{24}. A block from the middle of the file is read in. It will be cached

a) in the file's mobj

b) in the file system's mobj

c) someplace else

d) not at all

Quiz 5

A file is created and one byte is written to it at location 2^{24}. The file is mmapped read-write and shared. A thread accesses an integer from a page in the middle of the mapped region. The page will be cached

a) in the file's mobj

b) in the file system's mobj

c) someplace else

d) not at all

Usage Examples

```
for (j=0; j<jMax; j++) {
  for (i=0; i<iMax; i++) {
    sum += A[i][j];
  }
}

for (i=0; i<iMax; i++) {
  for (j=0; j<jMax; j++) {
    sum += A[i][j];
  }
}
```


Results

- 48k x 48k matrix

- ji loop
  - 37:00
- ij loop
  - 4:12

Providing Advice to the Kernel

- madvise(start, length, advice)
  - normal
  - sequential
  - random
  - will need
  - don't need
  - and others …

Results

- 48k x 48k matrix

- ji loop
  - 37:00 (normal)
  - 29:49 (sequential)
  - 38:01 (random)
- ij loop
  - 4:12 (normal)
  - 3:03 (sequential)
  - 4:15 (random)

Security Part 1

Concerns

- Problems with user-level code
- Problems with kernel code

Code Defensively

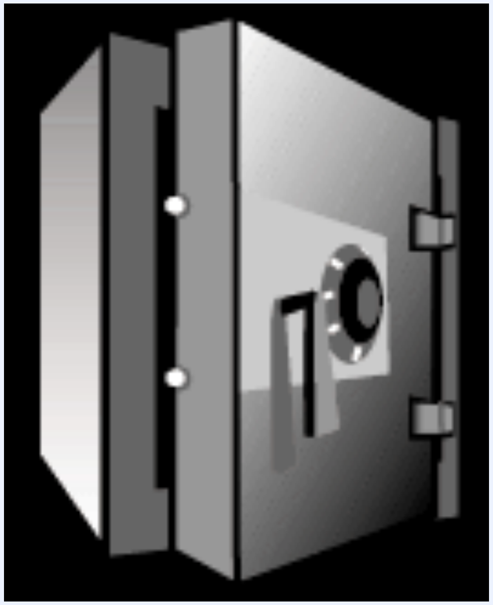
- Make sure your program does only what it's supposed to do
  - does the "right thing" for all possible sets of arguments
  - doesn't have weird (and unanticipated) interactions with other programs
- Particularly important if your program has "special privileges"

Change Roles

- It's more fun to play the attacker
- You can learn a lot by thinking through the attacker's role

Our System Is So Secure …

- … we challenge you to break in
  – come to our booth: anyone who breaks into our system gets $10,000



The System Didn't Survive …

Could I play with your system as an ordinary user?

Attacker

Marketing Person

Sure: it's secure so there's no problem

There may be a problem. Could you su as root and look at it?

types for a minute or so

Attacker

Marketing Person

Sure [su's as root] I'll run *ps* and see what's going on

Marketing Person

Everything looks fine to me [gets out of root]

[immediately becomes super-user] Give me my $10,000!

Attacker

What Happened

- **The attacker created, in the current directory, an executable file called *ps* containing:**

```
#!/bin/sh
cat >> /etc/passwd <!
bogus::0:0:root:/:/bin/sh
!
exec /bin/ps !*
```

- **The path variable in the root account was: ".:/usr/bin:/bin"**

Concerns

- Authentication
  - who are you?
- Access control
  - what are you allowed to do?
- Availability
  - can others keep you out?

Logging In …

- Username/password
  - who knows the passwords?

One-Way Functions

- *f(x)* is easy to compute
- *f* $^{-1}$*(x)* is extremely difficult, if not impossible, to compute
  - Unix password file contains image of each password
    - » /etc/passwd contains twd:y
    - » twd logs in, supplies x
    - » if *f(x)* == *y*, then ok
    - » /etc/passwd is readable by all

Dictionary Attack

- For all words in dictionary, compute *f(word)*
- Find *word* such that *f(word)* == *y*

Counterattack

- **Salt**
  - for each password, create random "salt" value
  - /etc/passwd contains *(f(append(word, salt)), salt)*
  - 12-bit salt values in Unix
  - attacker must do dictionary attack 4096 times, for each salt value
    - » done …
    - » Feldmeier and Karn produced list of 732,000 most common passwords concatenated with each of 4096 salt values
      - covers ~30% of all passwords

Counter Counter Attacks

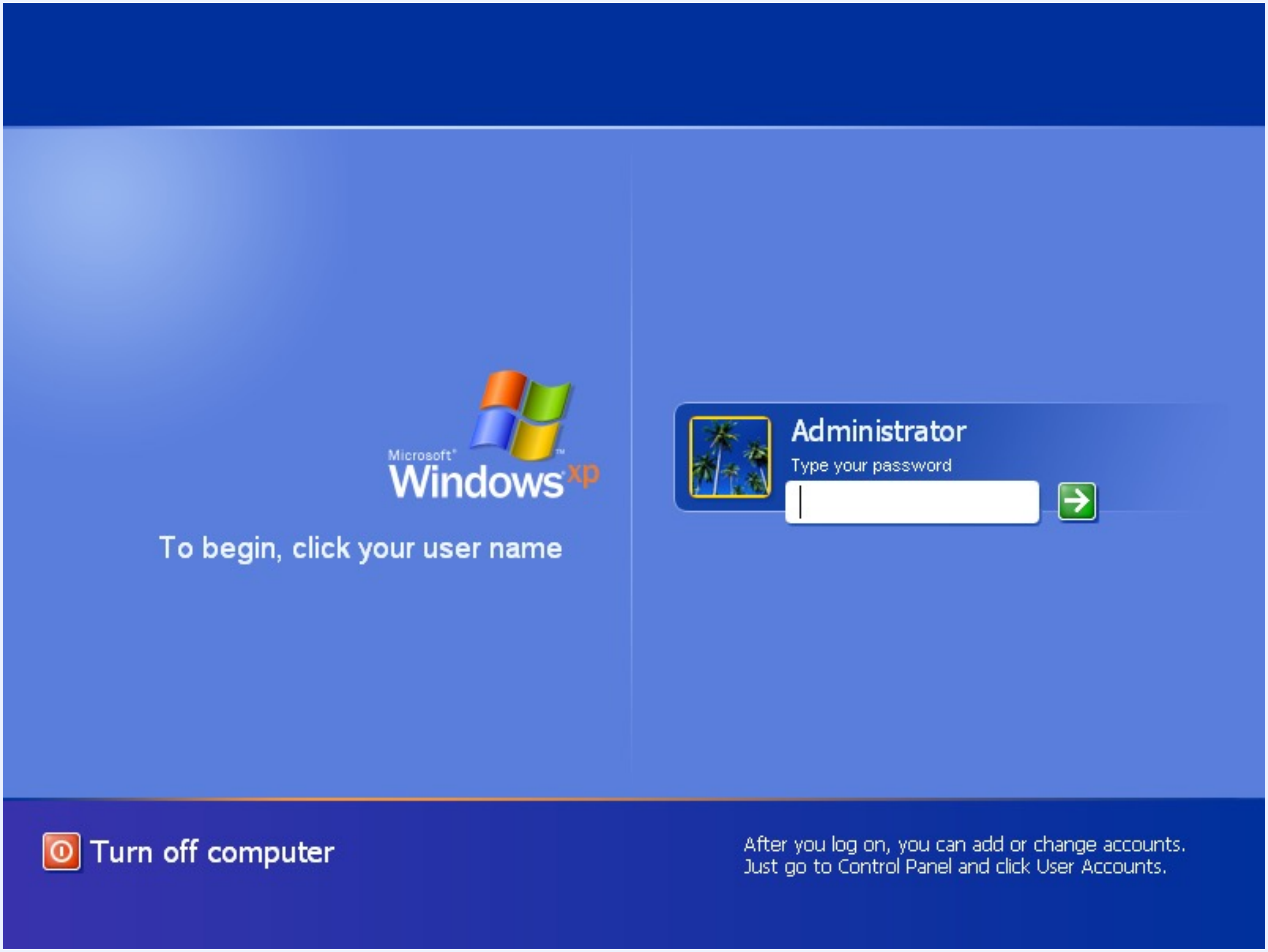
- Don’t allow common access to password images
  - /etc/passwd contains everything but password images and is readable by all
  - /etc/shadow contains password images, but is readable only by its owner
- Use better passwords
  - “w7%3nGibwy6” rather than “fido”
- Use strong cryptography and smart cards
  - combined with PINs
- Use two-factor authentication

Defeating Authentication

- **What are the prime factors of**

  5325138870287932192846843055513588820529482732761
  0407403175727513859436883214523893737052953027480
  7754890798107434809613388354335732832883202827204
  2055572159979867180328891700281777291005819624495
  2509309592137003269247211376423318797402174094174
  3851002617777645320194597739213700326924721137642
  3318797402174094174385100261777764532019459773388
  0145388493887041421512320698181588962921353458454
  9713993496308859388014538849388704142151232069818
  15889629213534584549713993496308859?

Defeat Authentication, Sneakily …



Quiz 6

Is there a useful defense against the attack in the previous slide?

a) no: it's utterly hopeless

b) yes, but it depends on users being educated (i.e., just somewhat hopeless)

c) yes, it works for even the naïve user

Hacking

- How to …
- Prevention

Attacks

- Trap doors
- Trojan horses
- Viruses and worms
- Exploit bugs
- Exploit features

Trap Doors

- You supply an SSD driver
- `ioctl(ssd_file_descriptor, 0x5309)`
  - standard command to eject the SSD
- `ioctl(ssd_file_descriptor, 0xe311)`
  - second argument is passed to your driver
  - on receipt, your driver sets UID of current process to zero

Trap-Door Prevention

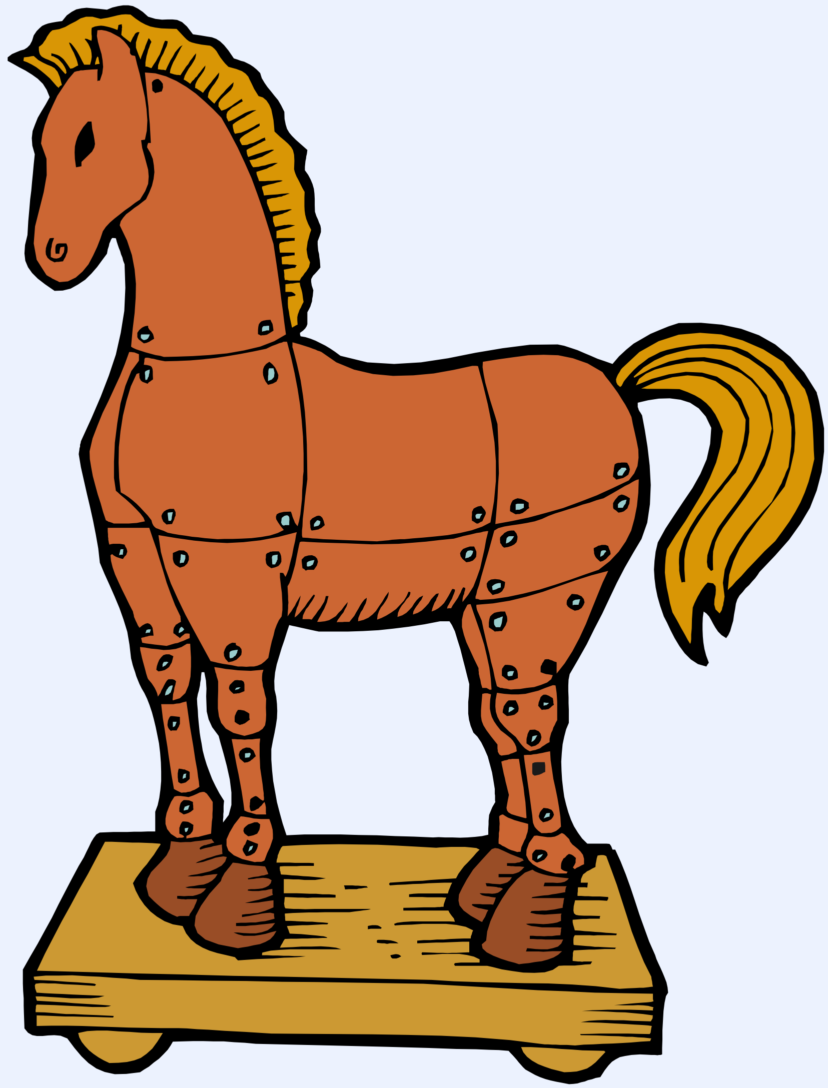
- Make sure everything that goes into kernel is ok
  - the Linux kernel has over 19,000 source-code files
  - also must worry about all setuid programs
  - Windows probably has more files
- How?
  - Windows
    - really careful management
  - Linux
    - thousands of eyes checking things out

Not Good Enough

- Paul Karger and Roger Schell
  - wrote 1974 paper suggesting compiler could add trap doors
- Ken Thompson was inspired and did it
  - his C compiler created a trap door in login program
  - C compiler added code to do this whenever it compiled itself
  - “feature” in C-compiler binary
  - self replicates — not in source code!
  - original source code deleted

Trojan Horses

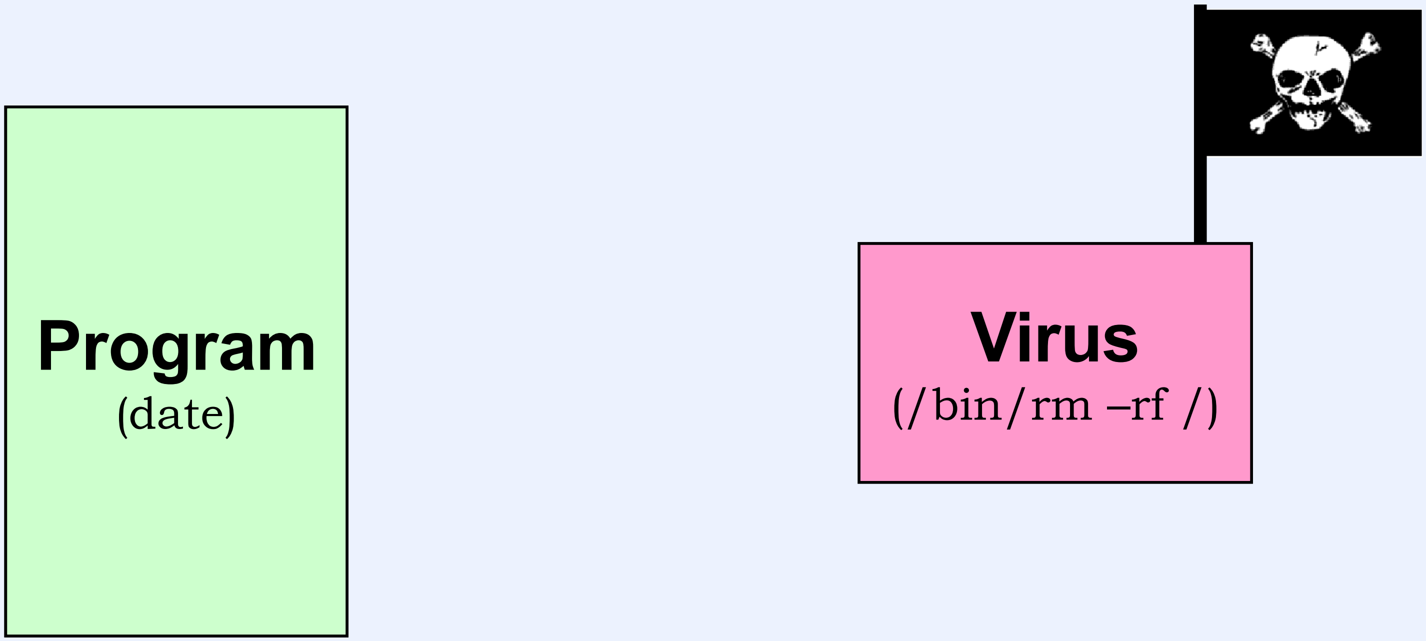
- **Free software!!!**
  - upgrades your four-core processor to eight-core!!



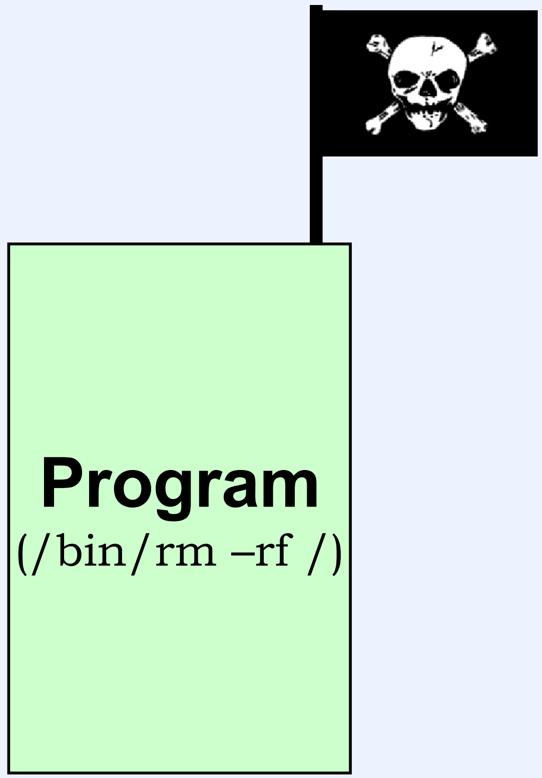
Viruses and Worms

- Virus: an “infection” of a program that replicates itself
- Worm: a standalone program that actively replicates itself

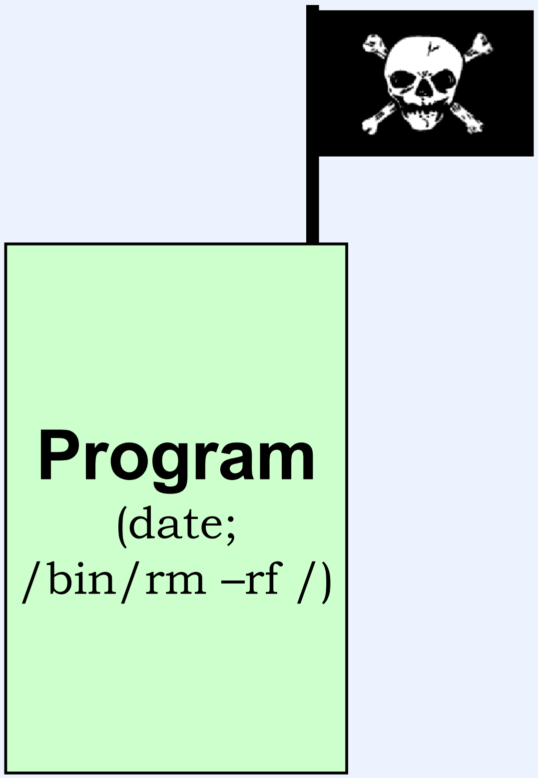
How to Write a Virus (1)



How to Write a Virus (2)



How to Write a Virus (3)



How to Write a Virus (4)

Program

(date;
if (day ==
Tuesday)
/bin/rm –rf /)

How to Write a Virus (5)

Program

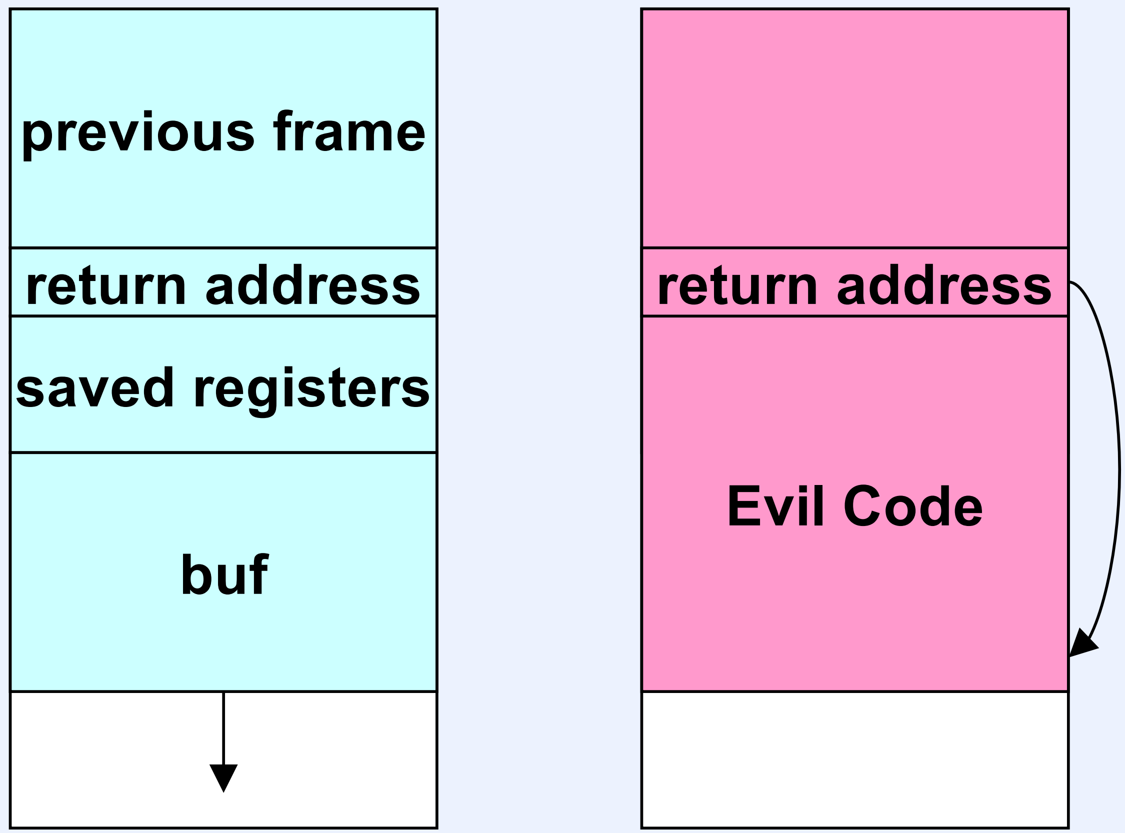
(date;
if (day ==
Tuesday)
/bin/rm –rf /;
infect
others)

Further Issues

- Make program appear unchanged
  - don’t change creation date
  - don’t change size
- How to infect others
  - email
  - web
  - direct attack
  - etc.

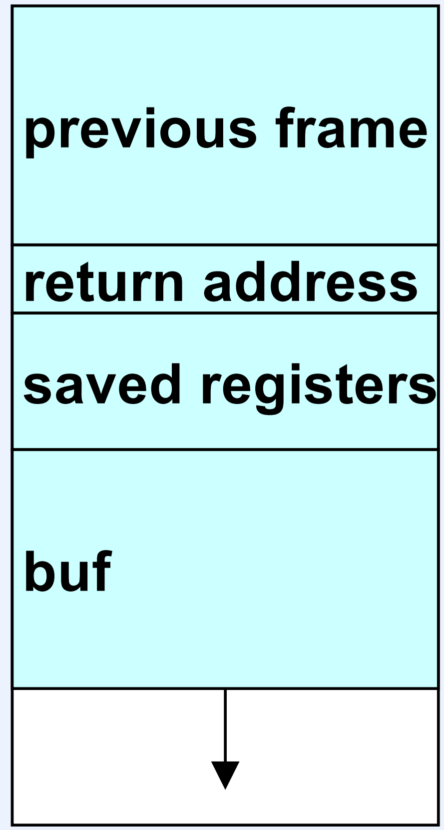
Buffer Overflow

```
void fingerd( ) {
    char buf[80];
    ...
    gets(buf);
    ...
}
```



Defense

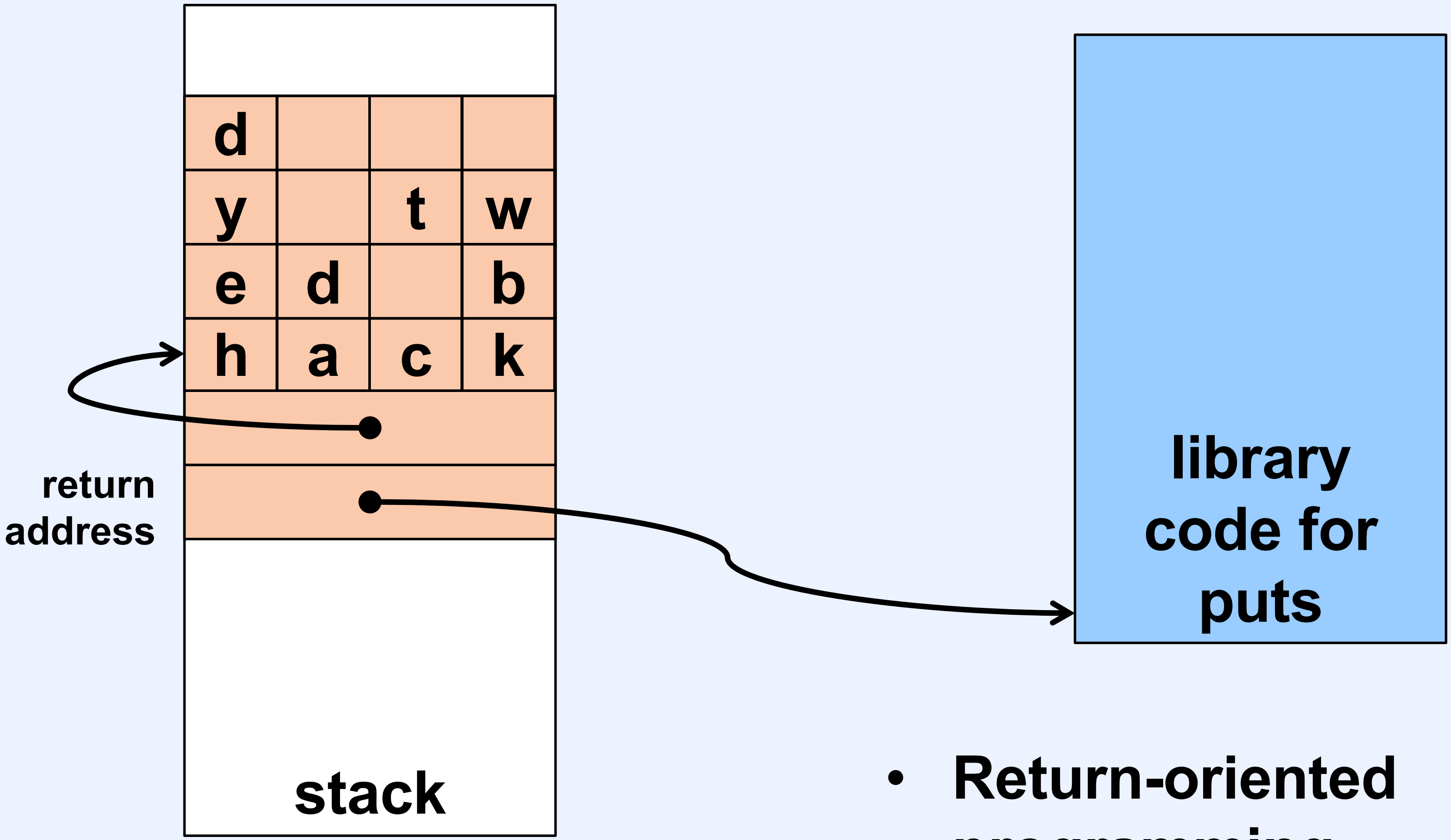
```
void proc( ) {
    char buf[80];
    ...
    fgets(buf, 80, stdin);
    ...
}
```



Better Defense

- Why should the stack contain executable code?
  – no reason whatsoever
- So, don't allow it
  – mark stack *non-executable*
    - (how come no one thought of this earlier?)
    - (Intel didn't support it till recently)
- Data execution prevention (DEP)
  – adopted by Windows and Linux in 2004
  – by Apple in 2006

Offense

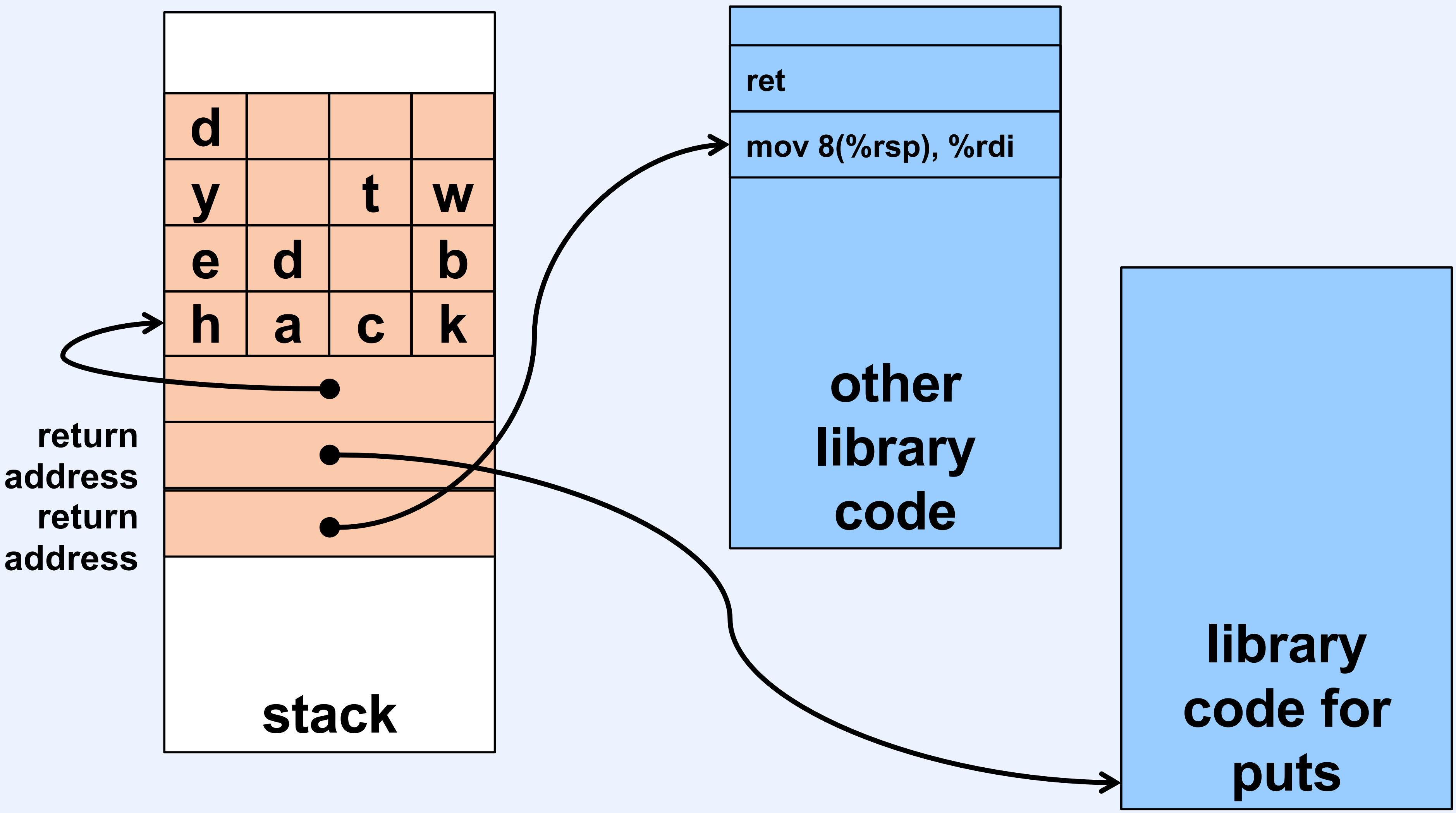


- Return-oriented programming

Defense

- Example assumes parameters passed on stack
  - 32-bit x86 convention
- Switch to x86-64
  - parameters passed in registers
  - example breaks
- Offense foiled?

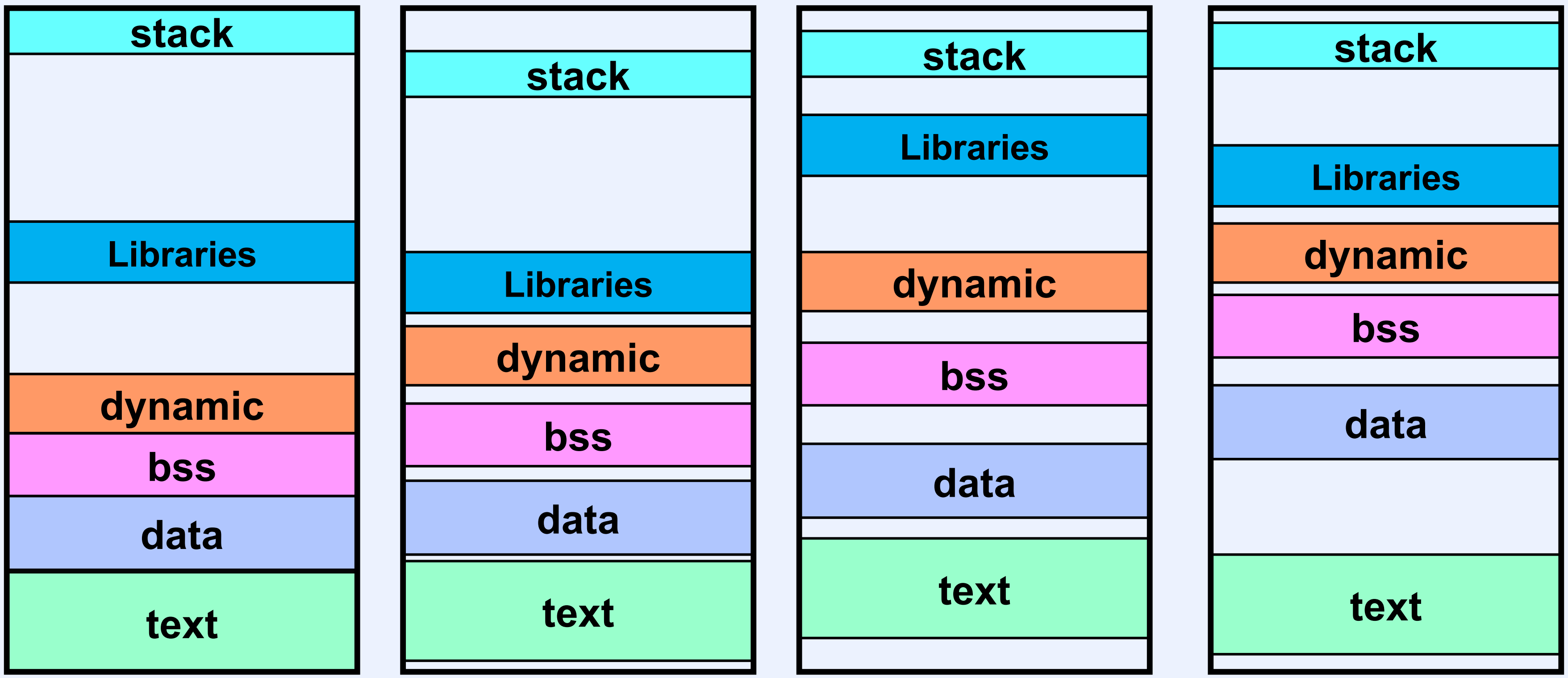
Offense



d
y
t
w
e
d
b
h
a
c
k
return address
return address
stack
ret
mov 8(%rsp), %rdi
other library code
library code for puts

Defense

- Address space layout randomization (ASLR)
  - start sections at unpredictable locations



Offense

- One possibility
  - guess the start address
    - perhaps $1/2^{16}$ chance of getting it right
    - repeat attack 100,000 times
      - won't be noticed on busy web server
      - very likely it will (eventually) work