# CSC 374/407: Computer Systems II

Lecture 6

Joseph Phillips
De Paul University

2016 November 28

# Reading

- Bryant & O'Hallaron "*Computer Systems, 2$^{nd}$ Ed.*"
  - Chapter 9.1-9.8: Virtual Memory
  - Chapter 9.9: Dynamic Memory Allocation
- Hoover "*System Programming*"
  - Chap 4: Pointers and Structures
    - Especially: 4.2.3: Dynamic Memory Allocation

# Topics

- Linux virtual memory and paging
- The heap
  - Heap Motivation
  - Heap Programming at C level (glibc)
    - malloc(), free(), calloc() and realloc()
    - How ***not*** to abuse the heap
  - Heap Programming at OS level (Linux)
    - getrlimit(), setrlimit(), brk(), sbrk()
- C-Strings
  - Buffer overflow attacks
  - Preventing buffer overflow attacks

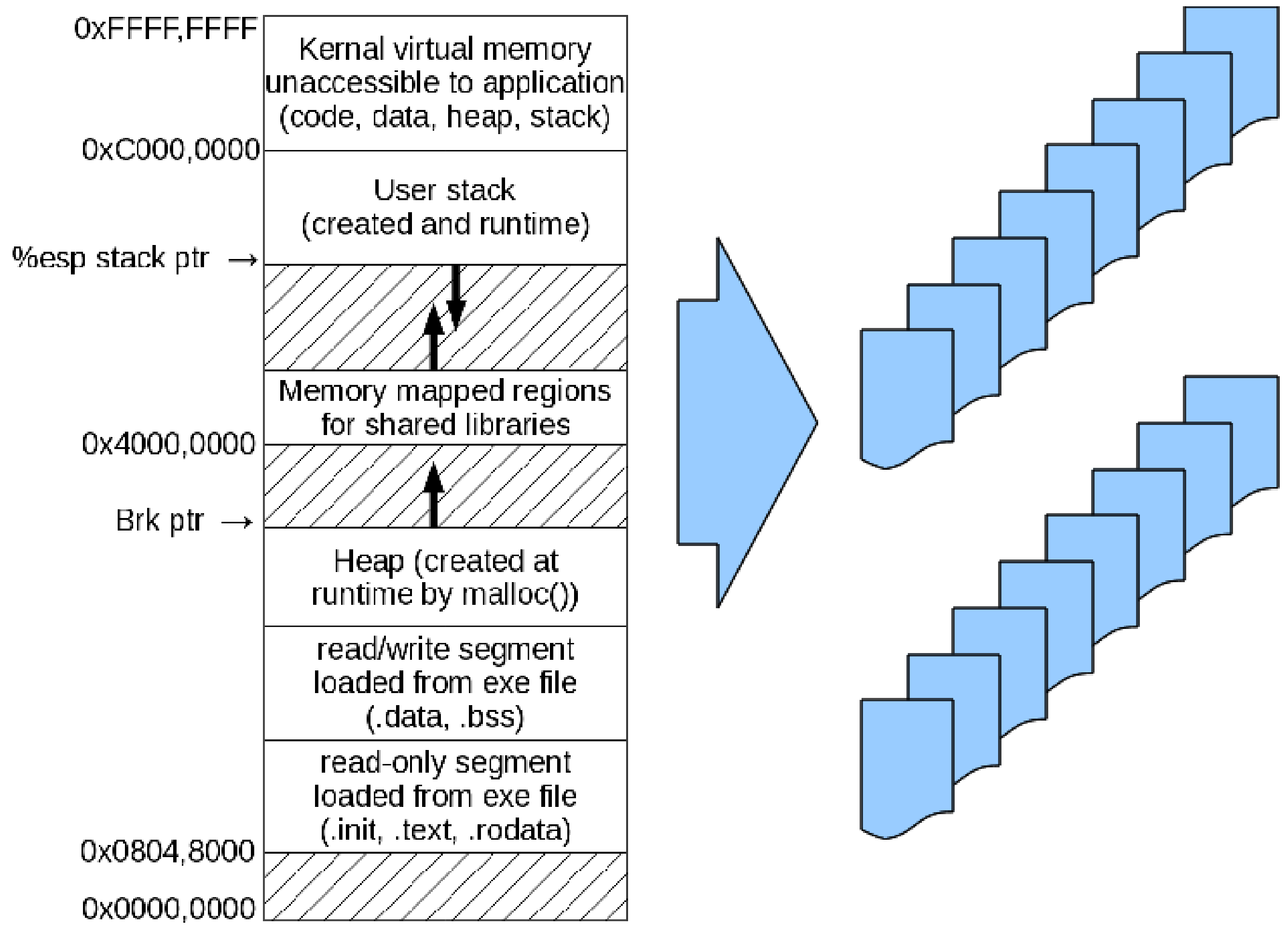
# Virtual Memory and Paging

- Advantages of virtual memory

1. Access to more memory than just Dynamic RAM (“DRAM”)
2. Easier memory management, let's processes share pages
3. Increased protection for a process' memory: either a process has access to a page or it does not.

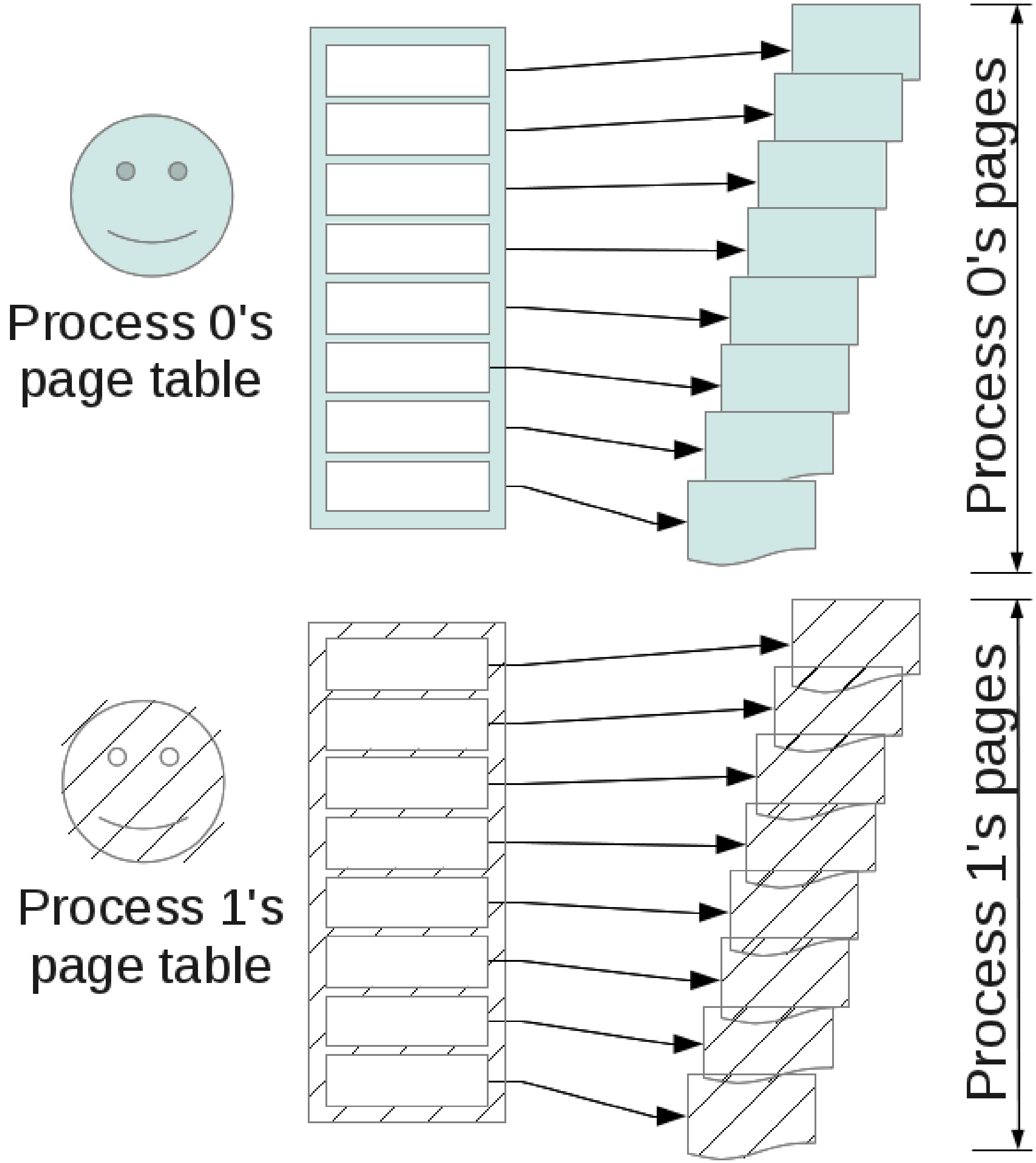
# What is virtual memory? (1)

- Each process' memory divided into pages.
- Pages 4 kb each.



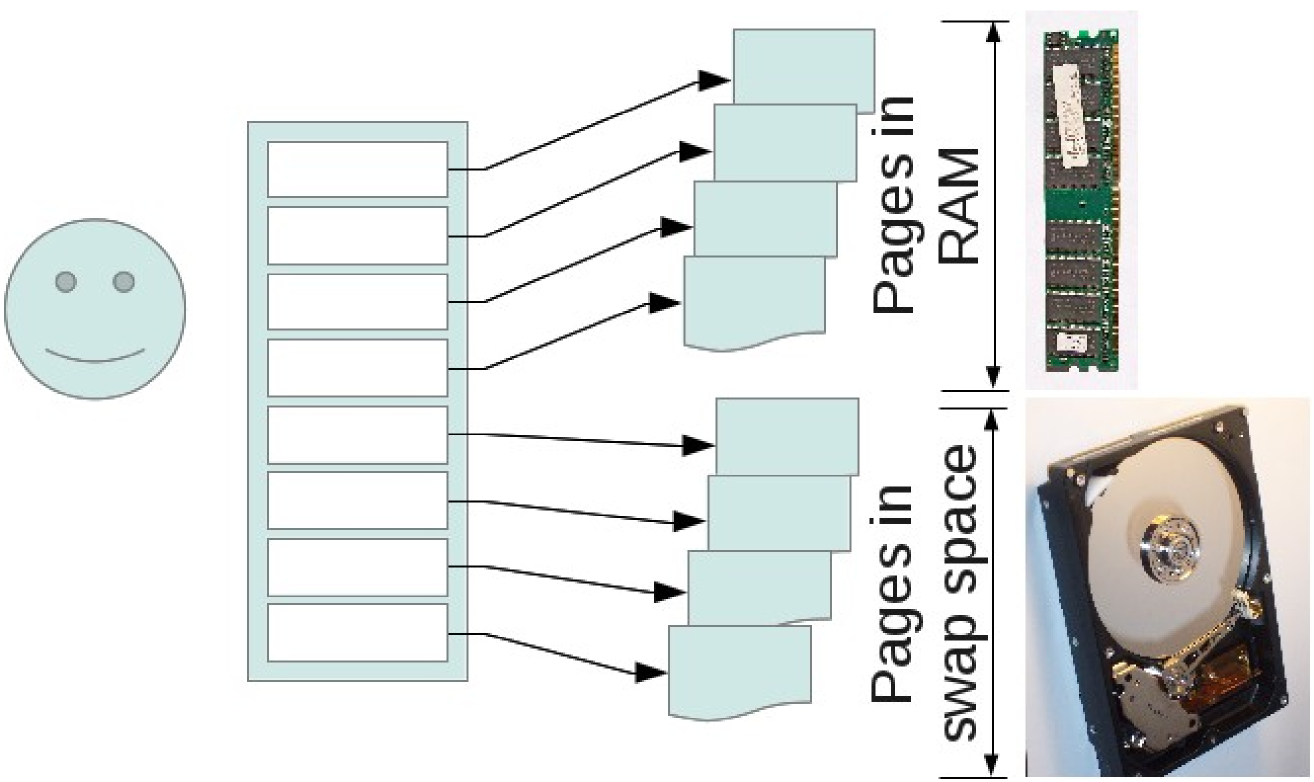
# What is virtual memory? (2)

- Each process' ***page table*** tells which pages it owns.



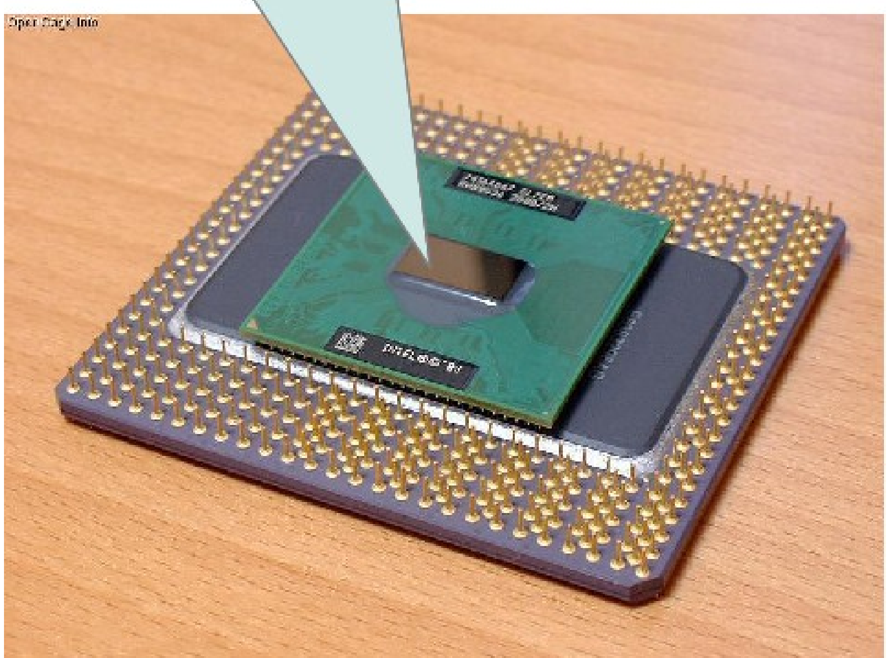
# What is virtual memory? (3)

- Not everyone's pages fit in memory at same time.
- Processes own ***virtual pages***, implemented either:
  - *as* ***physical pages*** (in RAM: ***FAST!***)
  - *as* ***swap space*** (on the harddrive: ***SLOW!***)



# The CPU thinks in terms of ***virtual addresses***

Give me the memory at address 0x1234,5678 please



- Your program knows ***virtual addresses*** in ***virtual memory***
- They have to be translated into addresses in ***physical memory***

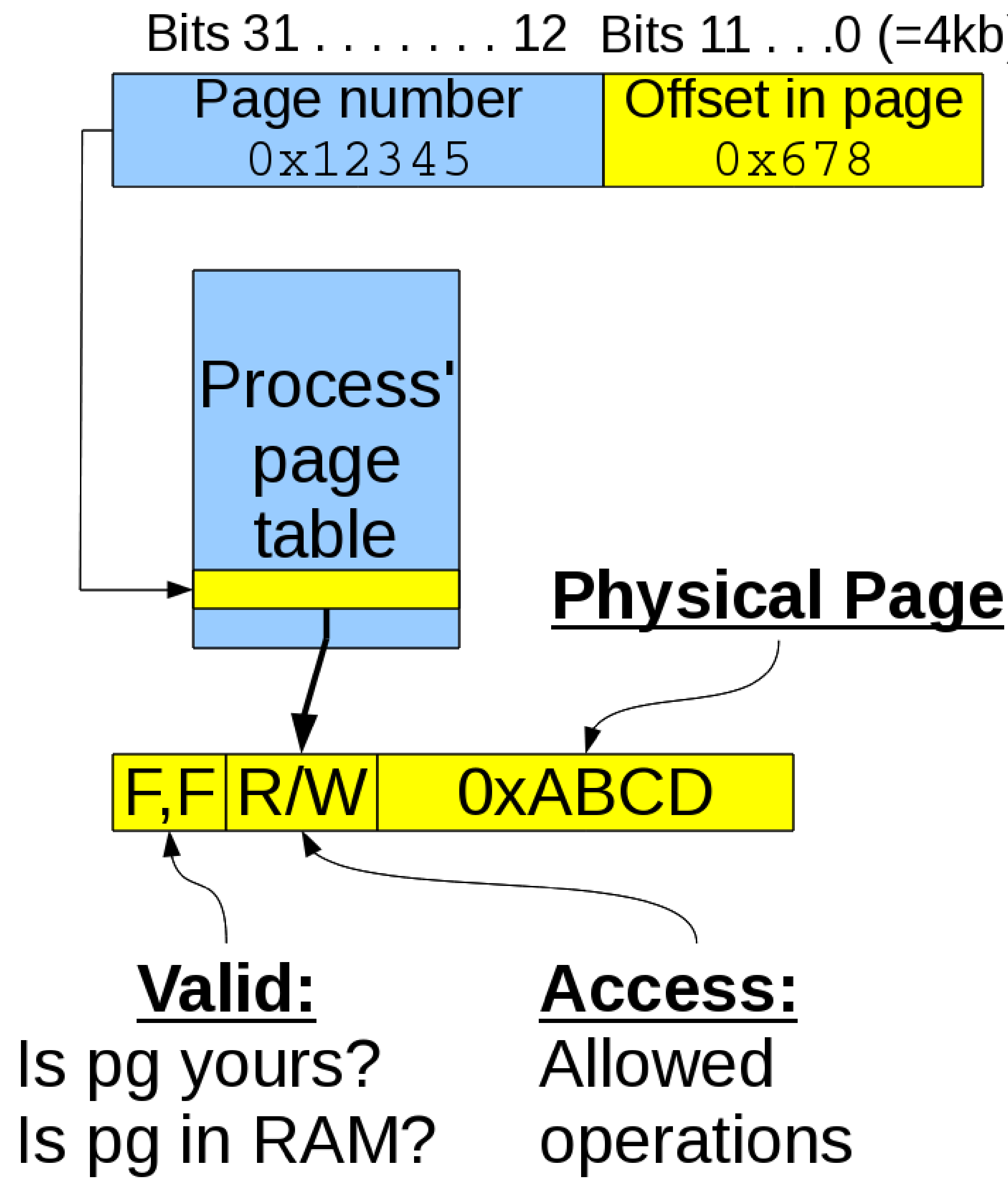
# Translating Virtual to Physical Addresses (32 bits) (1)

Each process has a page table.

High bits 12..31 tell the ***virtual page num.***

Low bits 0..11 tell the ***offset within page***.

Page table tells page's
- Validity
- Allowed access
- Physical page (or location on disk)

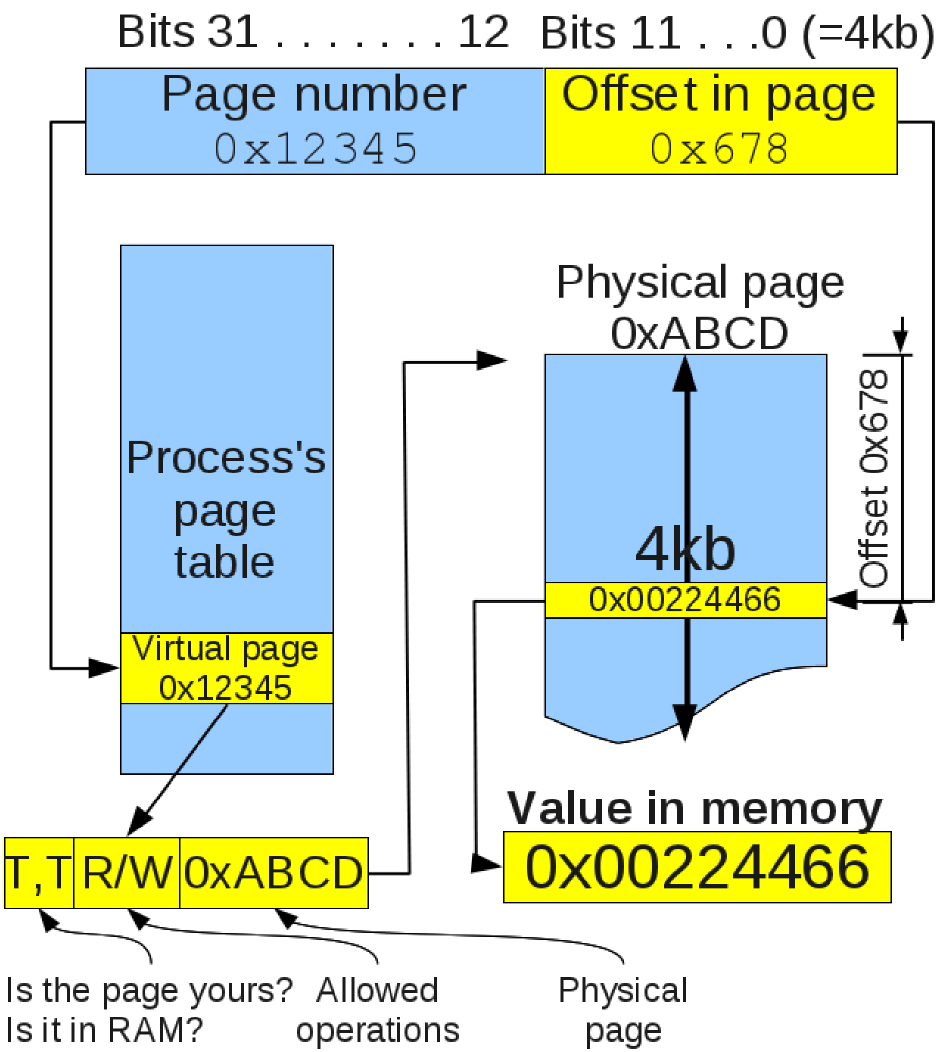


# Virtual Memory Operation 1

***Case 1***: If:

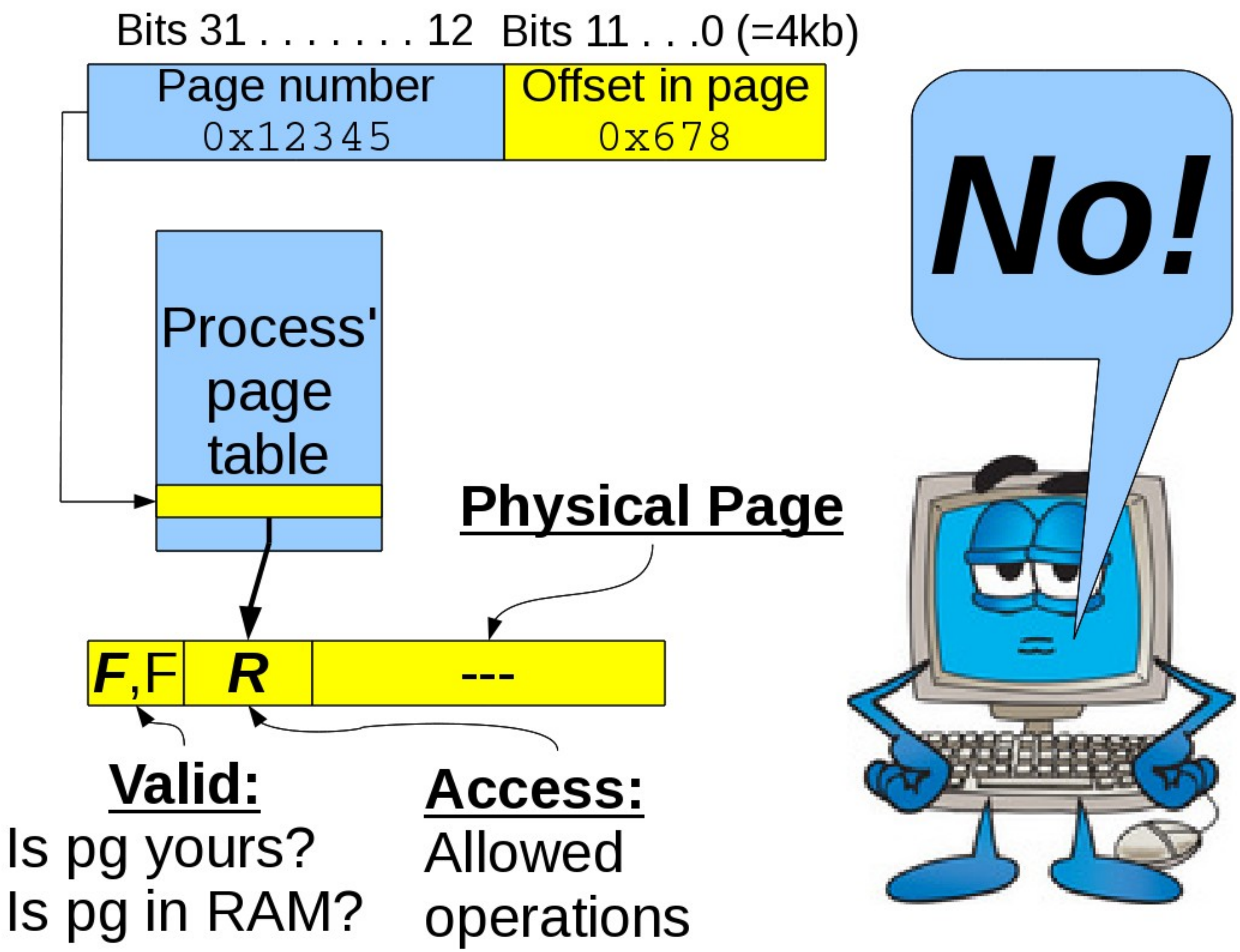
- Page belongs to process, AND
    - Page is in memory

then go to mem[physicalPage + offset]



# Virtual Memory Operation 2

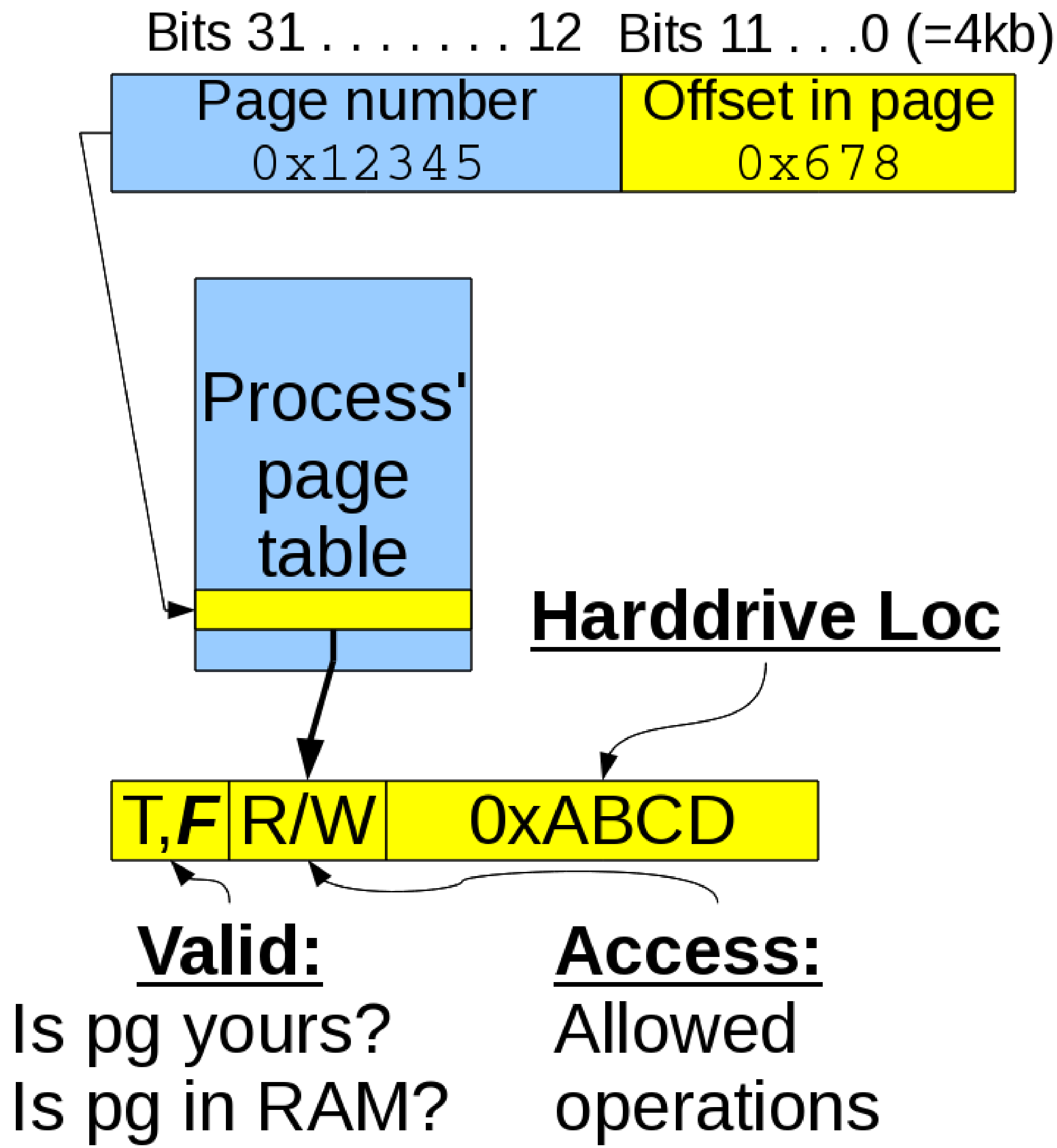
***Case 2***: You don't own the page, or try to write to read-only page



# Virtual Memory Operation 3

***Case 3***: Page is yours but not in RAM.

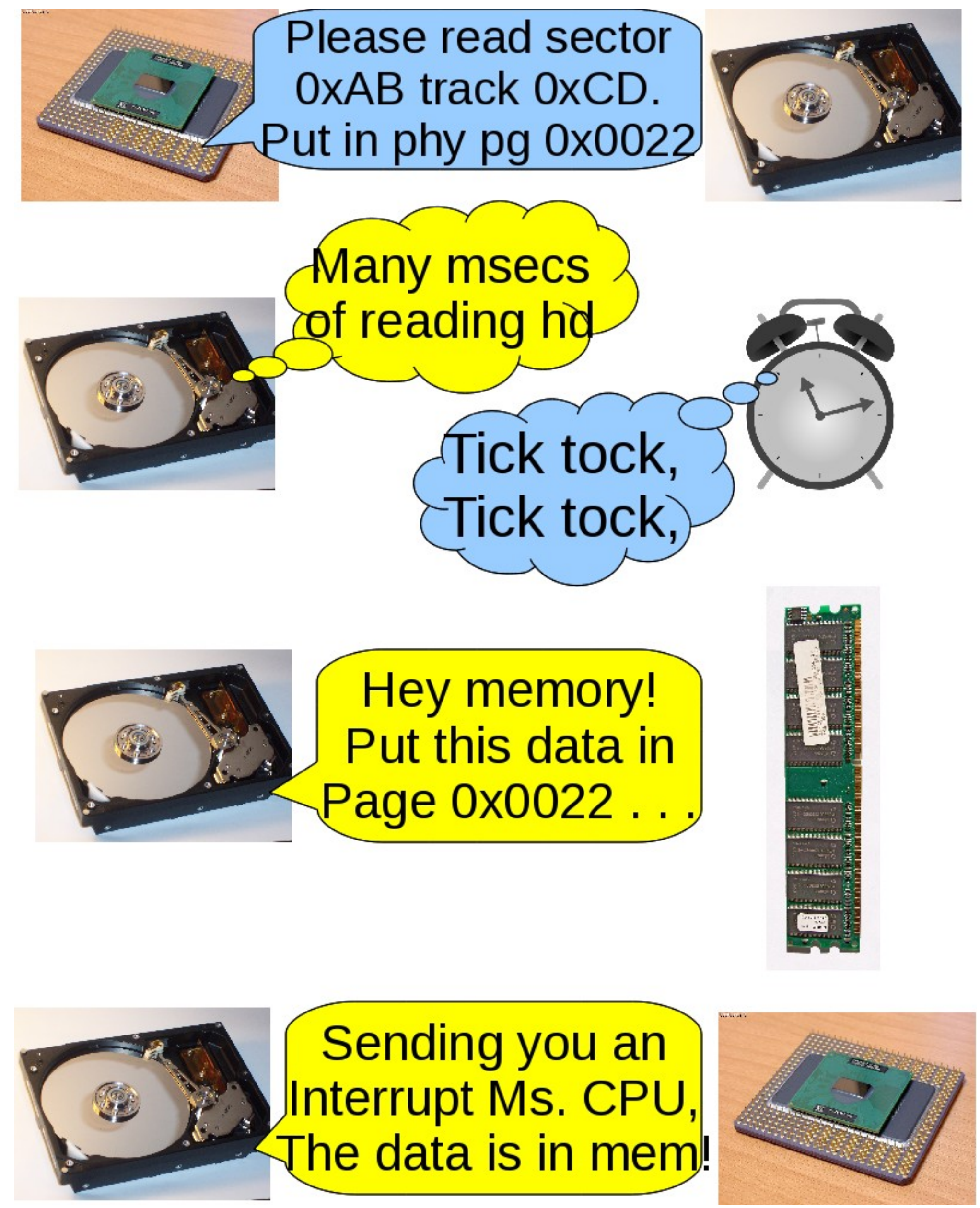
Have to load from hard drive.



# Virtual Memory Operation 3, cont'd

***Case 3***: Page is yours but not in RAM.

Harddrive (and network card, and flash drive, and DVD, *etc*.) writes data directly to RAM.  Interrupts CPU when done.

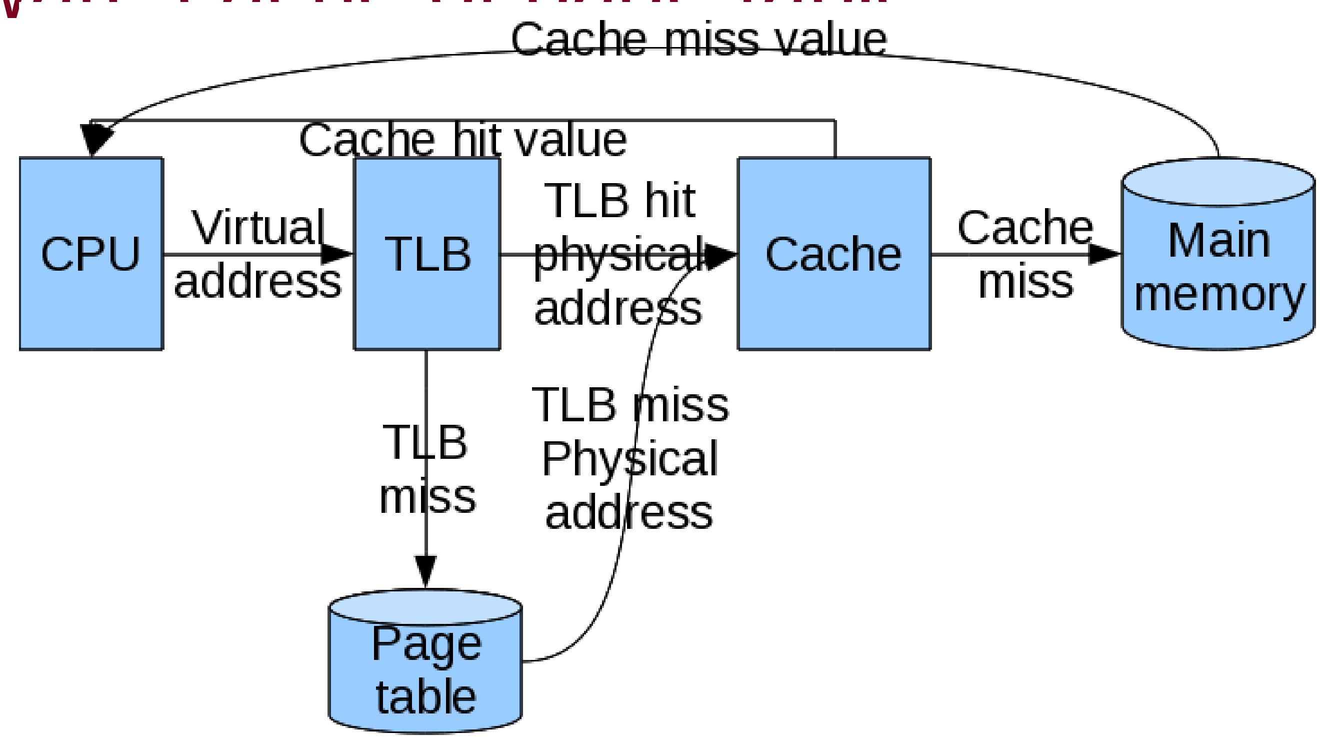


# Virtual Memory Operation 1 (more detail)

***Case 1***: Page is yours and is in RAM. Go to cache (or RAM) and get it!

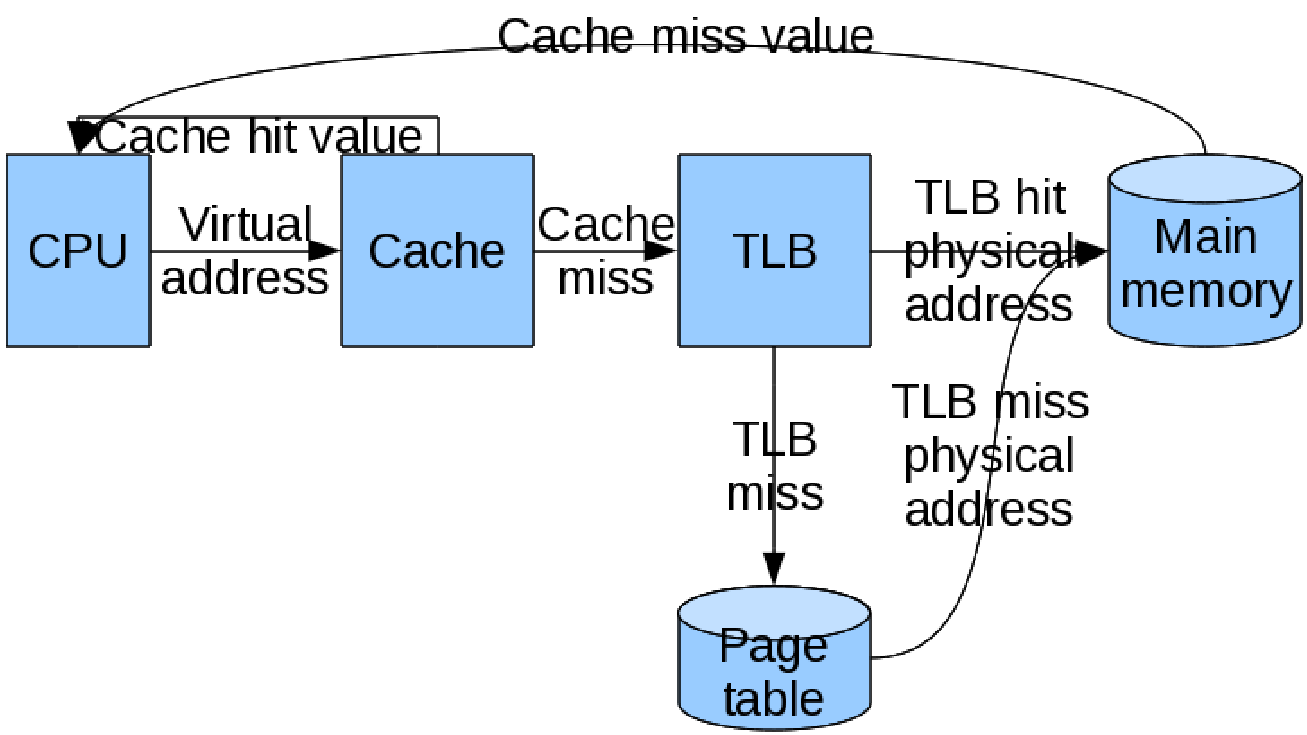
***Problem***: Page table could be huge!

***Solution***: TLB (Translation Lookaside Buffer) is a hardware cache of page table



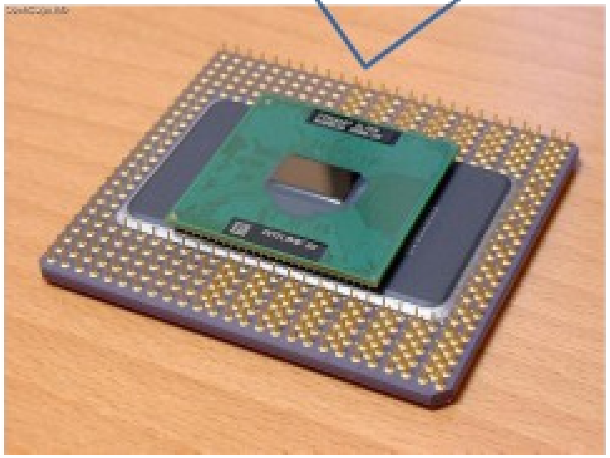
# Your turn!

You could put the ***cache*** before the ***TLB***. Caching ***virtual addresses*** would be ***faster*** for CPU. *Why is caching physical addresses still preferred?* **Hint**: There's more than 1 process.



# A single call to the TLB is okay for 32-bit, but what about *64-bit*?

Ms. CPU “*Please get me what’s at 0x1234,5678,1234,5678*”



TLB

# Then make multiple calls to TLB!

First iteration

TLB

*0x**1234**,5678,1234,5678*

Second iteration

TLB

*0x1234,**5678**,1234,5678*

Third iteration

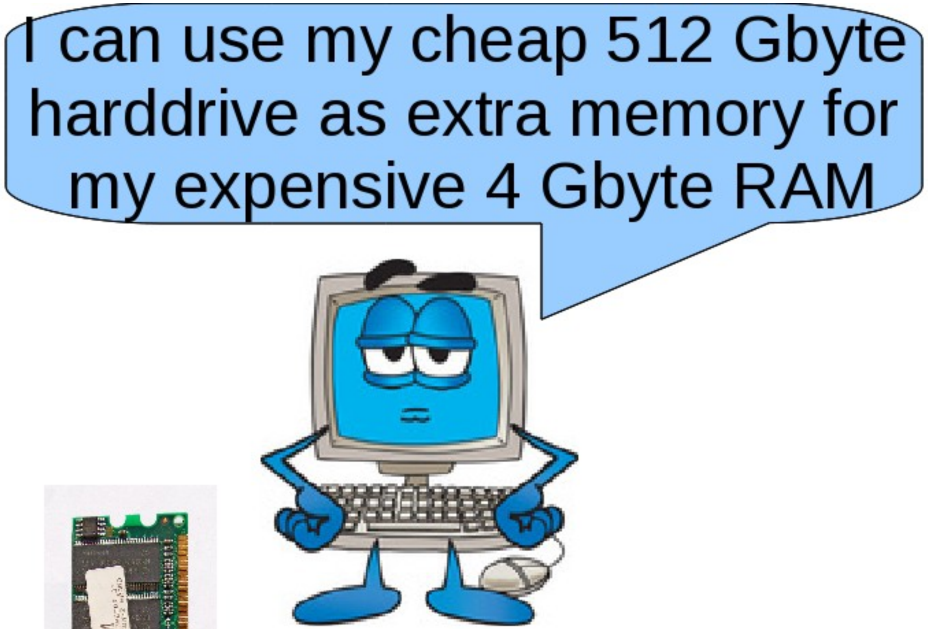
TLB

*0x1234,5678,**1234,5**678*

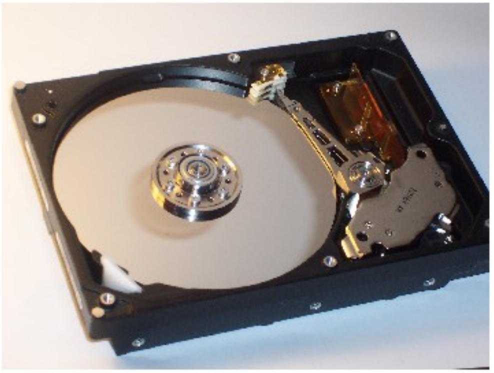
# Advantage #1 of Paging Sys

Ability to use harddrive as memory, not just file storage.

***Question***: Nothing is free! What did we sacrifice?




4 Gbyte expensive!

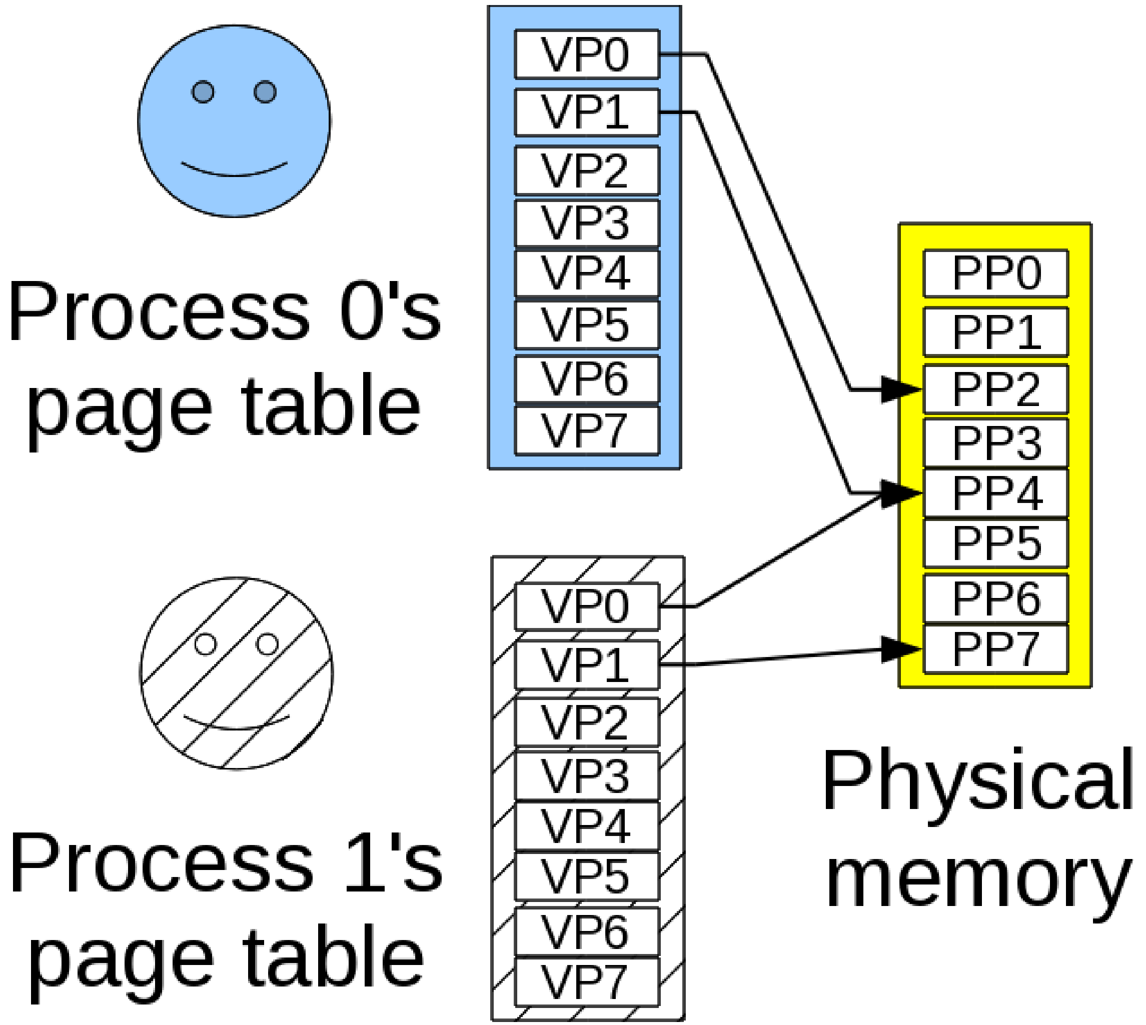


512 Gbyte, cheap!

# Advantage #2 of Paging Sys

Ability of processes to share pages, and flexibility for OS about where virtual pages are in phys mem.

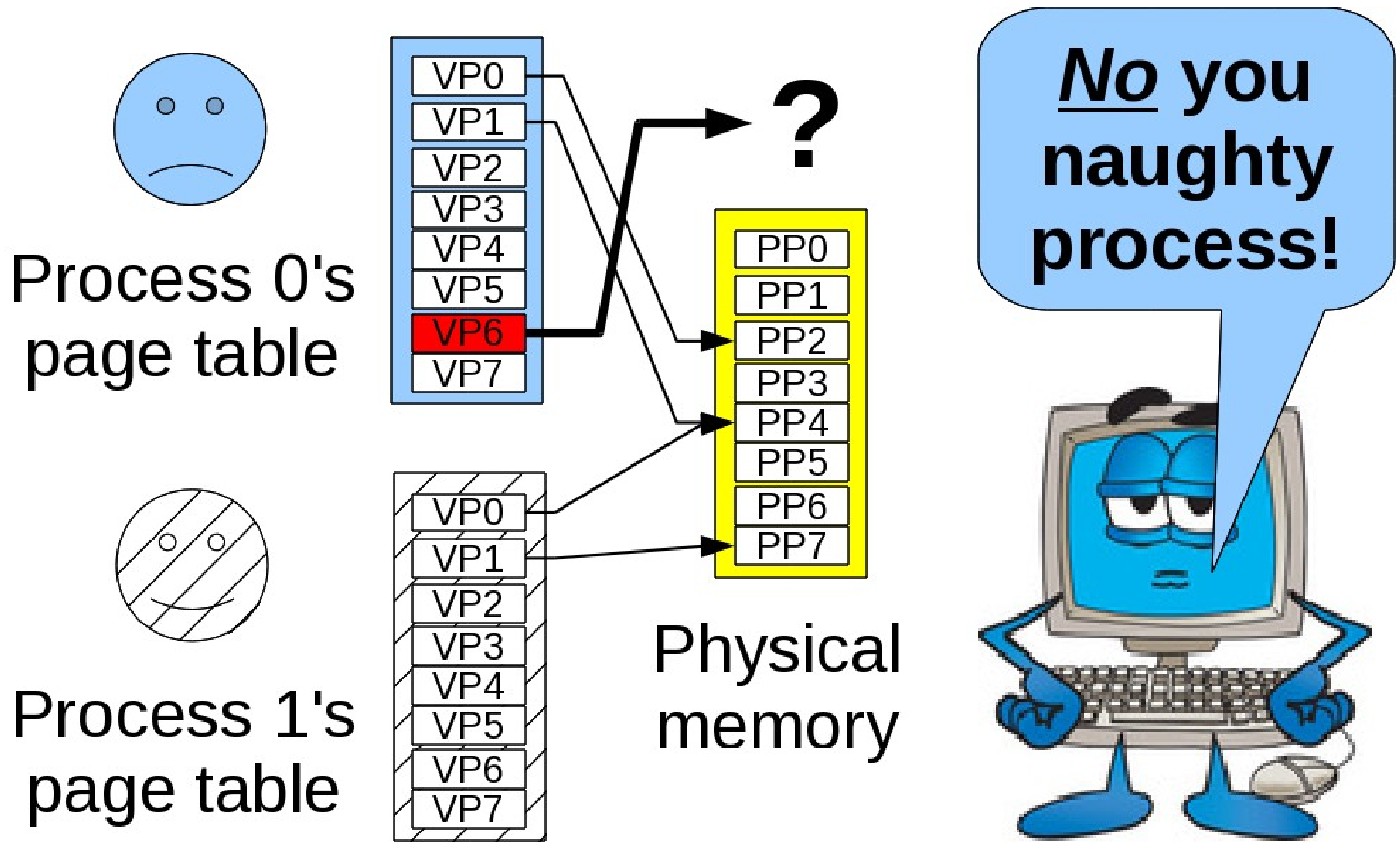
***Question***: Which segment pages can be safely shared?



# Advantage #3 of Paging Sys

Ability to protect processes from each other.

***Question***: What must the OS do when process makes illegal memory access?



# Effective main memory use

Stay on the same page!

# Your turn!

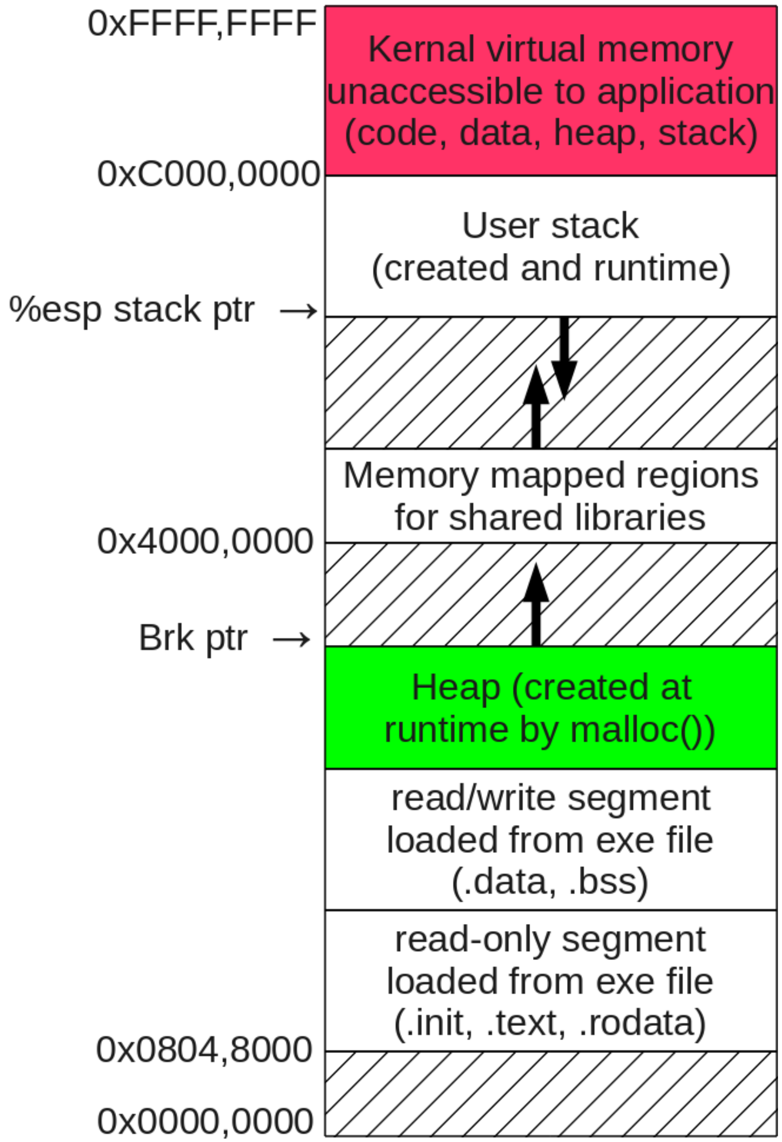
Which has better spatial locality?

```
// Option 1:
int sum=0;
for (int i=0; i<NUM_ROW; i++)
  for (int j=0; j<NUM_COLS; j++)
    sum+=array[i][j];

// Option 2:
int sum=0;
for (int j=0; j<NUM_COLS; j++)
  for (int i=0; i<NUM_ROW; i++)
    sum+=array[i][j];
```

# Today's topic (in space)

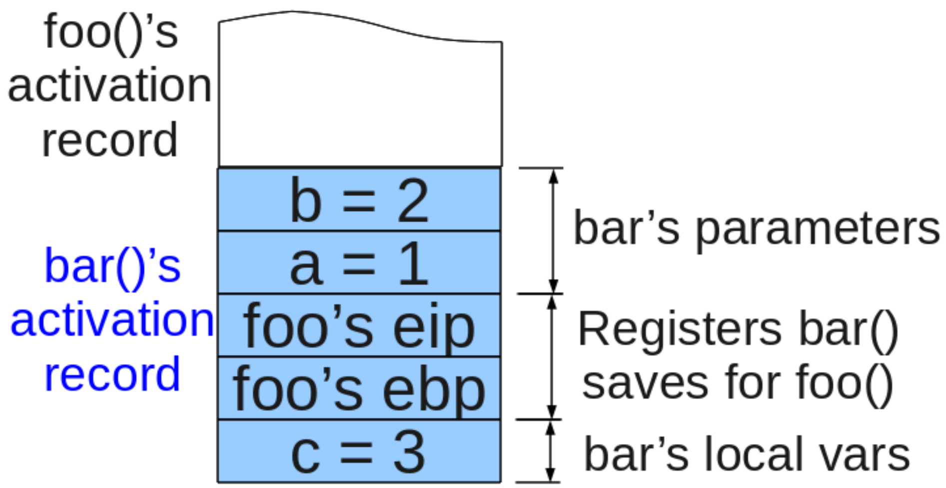
- Runtime heap



# We all know that local vars live on the stack

```
int bar (int a, int b)
{
   int c = a + b;
   return(c);
}

int foo ()
{
   int x = bar(1,2);
   . . .
```

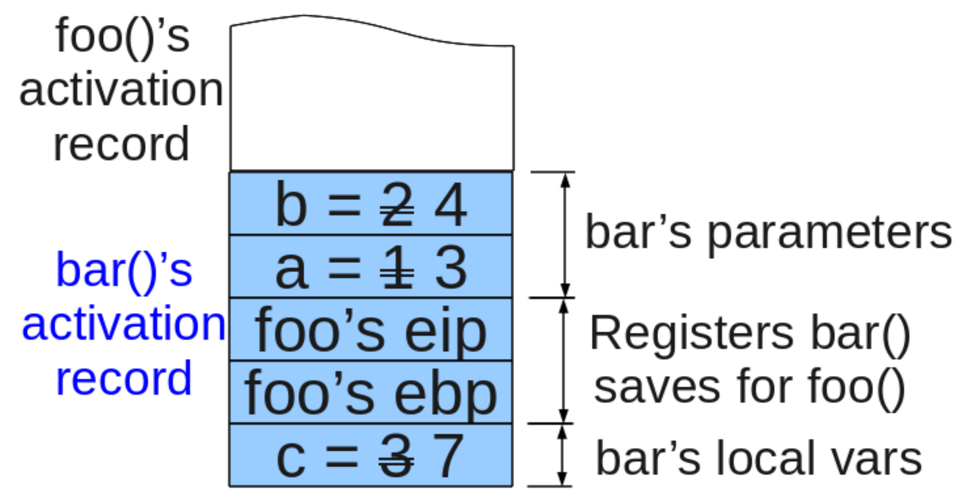


# And we all know that the stack gets overwritten by subsequent function calls

```
int bar (int a, int b)
{
   int c = a + b;
   return(c);
}

int foo ()
{
   int x = bar(1,2);

   x = bar(3,4);
   . . .
```



# Question?

But what if you
want your object to
*persist?*

# Answer!

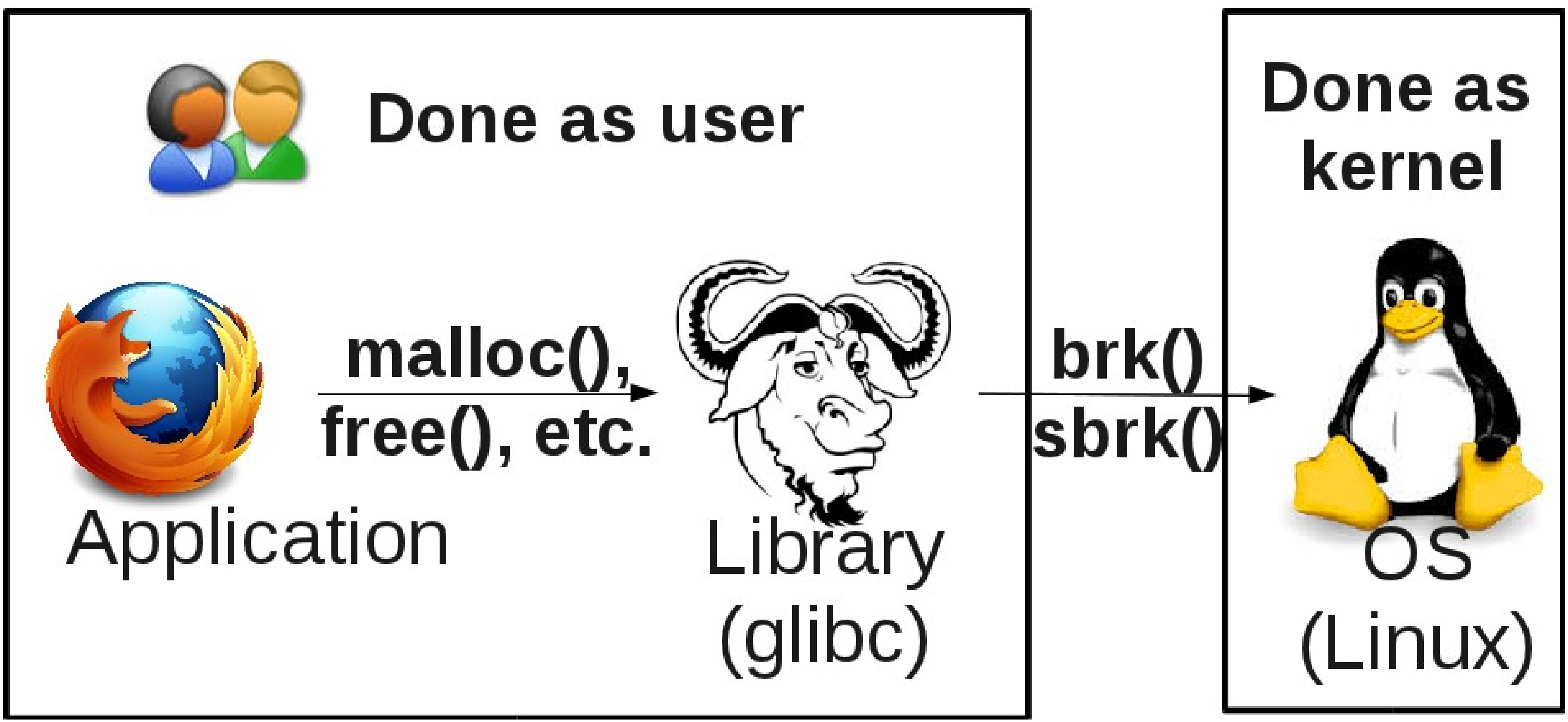
Put it on the ***heap!***

```
MyClass* bar()
{
  return new MyClass(1,2);
}

void foo()
{
  MyClass ptr;

  ptr = bar();
  . . .
  delete(ptr);
```

# Two “folks” we have to ask for heap memory:



# First: Ordinary C heap programming (glibc)

# Heap Programming

- ***void* malloc(size_t numBytes)***
  - Allocate ***numBytes*** bytes from heap and return pointer (allocates in ***page table***, not necessarily ***main memory!***)
  - Returns ***NULL*** if error like not enough space.
  - Returned pointer should be cast to a more specific type before using:

```
int*  intPtr  = (int*)malloc(sizeof(int));
char* charPtr = (char*)malloc(strlen(sourcePtr)+1);
```

- ***void free (void* ptr)***
  - Return memory back to system

# malloc() and free() example

```
#include <stdlib.h>
#include <stdio.h>

int     main    ()
{
  int*  iPtr;

  iPtr  = (int*)malloc(sizeof(int));
  *iPtr = 14;
  printf("iPtr = %p, *iPtr = %d\n",
         iPtr,*iPtr
        );
  free(iPtr);   // Very important!
  return(EXIT_SUCCESS);
}
```

# realloc() and calloc()

- ***void* calloc(size_t nmemb, size_t size);***
  - Allocates an array of ***nmemb*** members, each of ***size*** bytes.
  - Initializes memory to byte all 0's
  - (Some claim this wastes time.)
- ***void* realloc(void *ptr, size_t size);***
  - "Re-allocates" by extending memory allocated at pointer, or by (1) getting ***size*** bytes, (2) copying from ***ptr*** into new memory, and (3) ***free()***-ing old memory

# realloc() and calloc() example

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_ELE 4
int      main    ()
{
  int*   iPtr;
  int*   iPtr2;
  int    i;

  iPtr  = (int*)calloc(NUM_ELE,sizeof(int));
  iPtr2 = (int*)calloc(NUM_ELE,sizeof(int));

  for  (i = 0;  i < NUM_ELE;  i++)
  { // Any other ways to access?
    iPtr[i] = i*10;
  }
```

# realloc() and calloc(), contd

```
  for  (i = 0;  i < NUM_ELE;  i++)
    printf("%3d is at %p\n",
           *(iPtr+i),(iPtr+i)
          );
  printf("Uh-oh, not enough space!\n");
 iPtr=(int*)realloc(iPtr,4*NUM_ELE*sizeof(int));

  for  (i = 0;  i < 4*NUM_ELE;  i++)
    printf("%3d is at %p\n",
           *(iPtr+i),(iPtr+i)
          );

  free(iPtr2);  // Very important!
  free(iPtr);   // Very important!
  return(EXIT_SUCCESS);
}
```

# realloc() and calloc(), contd

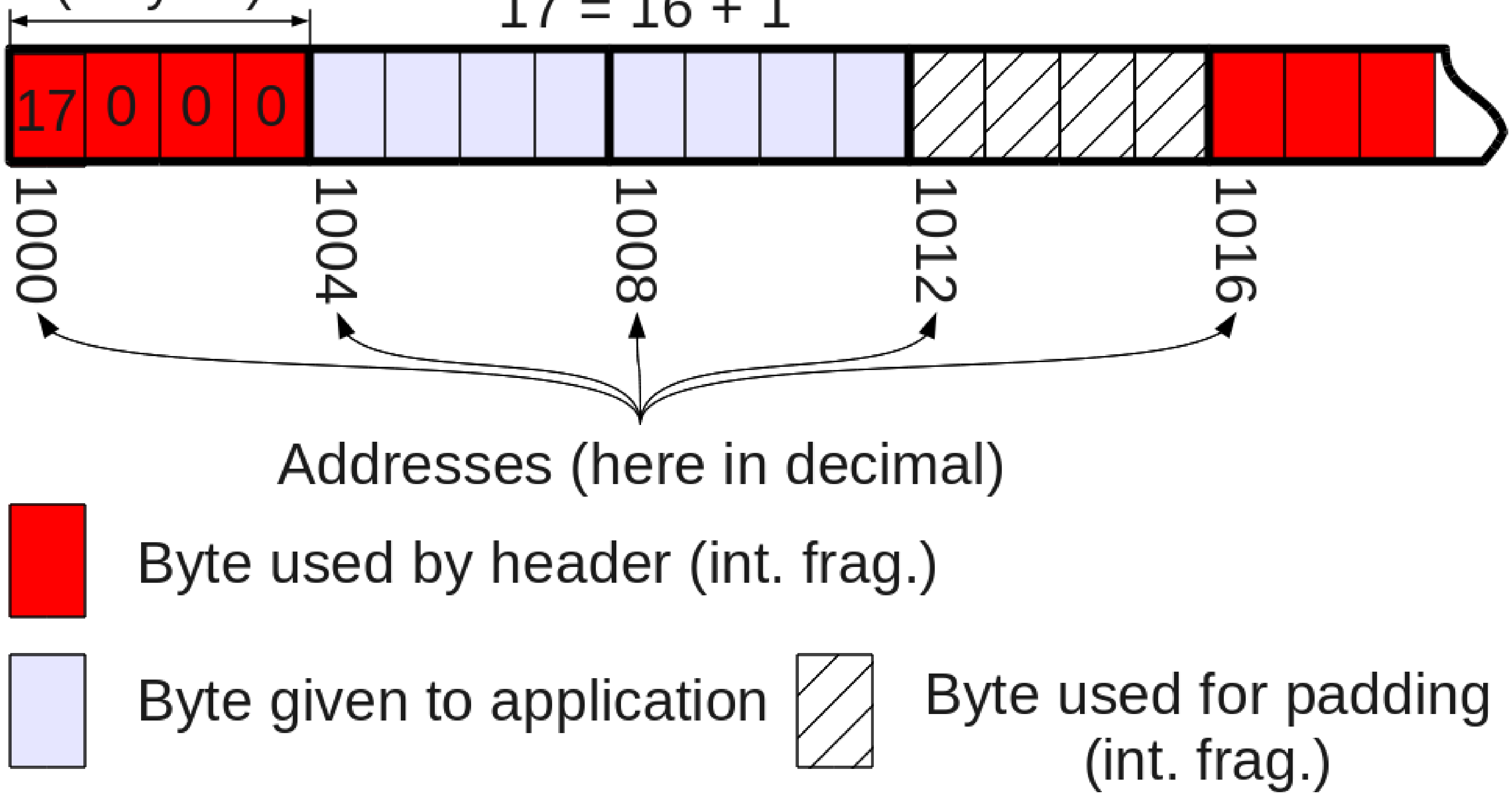
```
[instructor@localhost Lecture06]$ ./reallocAndCalloc1
  0 is at 0x84f2008
 10 is at 0x84f200c
 20 is at 0x84f2010
 30 is at 0x84f2014
Uh-oh, not enough space!
  0 is at 0x84f2038
 10 is at 0x84f203c
 20 is at 0x84f2040
 30 is at 0x84f2044
  0 is at 0x84f2048
  0 is at 0x84f204c
  0 is at 0x84f2050
  0 is at 0x84f2054
  0 is at 0x84f2058
  0 is at 0x84f205c
  0 is at 0x84f2060
  0 is at 0x84f2064
  0 is at 0x84f2068
  0 is at 0x84f206c
  0 is at 0x84f2070
  0 is at 0x84f2074
```

# Datastructure for heap management

(1) App. does ***malloc(6)***
(2) 6 rounded up to 8
(3) App. gets addr. 1004, block starts @ 1000

1 machine word
(4 bytes)

Header # = (block len) + [0 (free) | 1 (alloc)]
17 = 16 + 1



# Algorithm for heap management

- How do we keep track of malloc()s and free()s in heap?
- Desireable algorithms should:

1) Be efficient (maximize space for app. data)
    1) Minimize internal fragmentation (space used in allocated block for bookkeeping and padding)
    2) Minimize external fragmentation (unused space between allocated blocks)
2) Be fast!
3) Be robust (detect app errors)
4) Be sensitive to spatial locality (put similar sized items “close” to each other)
5) Be thread safe

# Your turn!

- Question 1:
  - Why do you think it changed addresses between the calloc() and the realloc()?
  - Can you test that hypothesis?
- Question 2:
  - Do you think the newly allocated space is guaranteed to by initialized to 0?
  - Can you test that hypothesis?

# A Rogue's Gallery of the *All-Time WORST* memory offenses

***Twirl your mustaches and enter at your own risk!***

# How NOT to use the heap: Memory leak

```
// Very subtly wrong.  What are the symptoms?
#include <stdlib.h>
#include <stdio.h>

int     main    ()
{
  int*  iPtr;

  iPtr  = (int*)malloc(sizeof(int));
  *iPtr = 14;
  printf("iPtr = %p, *iPtr = %d\n",
         iPtr,*iPtr
        );
  return(EXIT_SUCCESS);
}
```

# How NOT to use the heap: Double free

```
#include <stdlib.h>
#include <stdio.h>

int     main    ()
{
  int*  iPtr;

  iPtr  = (int*)malloc(sizeof(int));
  *iPtr = 14;
  printf("iPtr = %p, *iPtr = %d\n",
         iPtr,*iPtr
        );
  free(iPtr);
  free(iPtr);
  return(EXIT_SUCCESS);
}
```

# How NOT to use the heap: Wild-write 1

```
#include <stdlib.h>
#include <stdio.h>

int     main    ()
{
  int*  iPtr;

  iPtr      = (int*)malloc(sizeof(int));
  *(iPtr-1) = 14;
  printf("iPtr = %p, *iPtr = %d\n",
         iPtr,*iPtr
        );
  free(iPtr);
  return(EXIT_SUCCESS);
}
```

# How NOT to use the heap: Freed memory access

```
#include <stdlib.h>
#include <stdio.h>

int     main    ()
{
  int*  iPtr;

  iPtr  = (int*)malloc(sizeof(int));
  *iPtr = 14;
  printf("iPtr = %p, *iPtr = %d\n",
         iPtr,*iPtr
        );
  free(iPtr);
  (*iPtr)++;  // Not yet finished
  return(EXIT_SUCCESS);
}
```

# How NOT to use the heap: Uninitialized mem access

```
#include <stdlib.h>
#include <stdio.h>

int     main    ()
{
  int*  iPtr;

  *iPtr = 14;
  printf("iPtr = %p, *iPtr = %d\n",
        iPtr,*iPtr
       );
  return(EXIT_SUCCESS);
}
```

# This is how glibc complains:

```
*** glibc detected *** wildwrite: munmap_chunk():
invalid pointer: 0x08f71008 ***
======= Backtrace: =========
/lib/libc.so.6[0x49b8b9f2]
/lib/libc.so.6[0x49b8bc6b]
wildwrite[0x8048524]
/lib/libc.so.6(__libc_start_main+0xf3)
[0x49b2c6b3]
wildwrite[0x8048441]
======= Memory map: ========
08048000-08049000 r-xp 00000000 fd:03 397997
     ./wildwrite
08049000-0804a000 rw-p 00000000 fd:03 397997
     ./wildwrite
08f71000-08f92000 rw-p 00000000 00:00 0    [heap]
. . .
```

# And this is how Linux complains:

Segmentation fault (core dumped)

# Your turn!

- Fix the following function that should copy a string into memory allocated from the heap:

```
char* naughtyCopy (const char* fromP)
{
  char* toP;

  for  ( ;  *fromP != '\0'; fromP++, toP++)
    *toP = *fromP;

  free(fromP);
  return(toP);
}
```

# Second: OS's heap interface

# OS's heap tools

- A process can put limits on how big it's heap is allowed to grow
  - **`getrlimit()`**, **`setrlimit()`**
  - **Soft-limit**: limits how much heap (or stack, CPU, *etc*.) a process is allowed to use
    - Any process can set
  - **YOUR TURN:** Why would a process ***want*** to do this?
  - **Hard-limit**: limits the soft limit
    - Only privileged processes may set
- A process can get heap pages directly from OS:
  - **`brk()`**, **`sbrk()`**

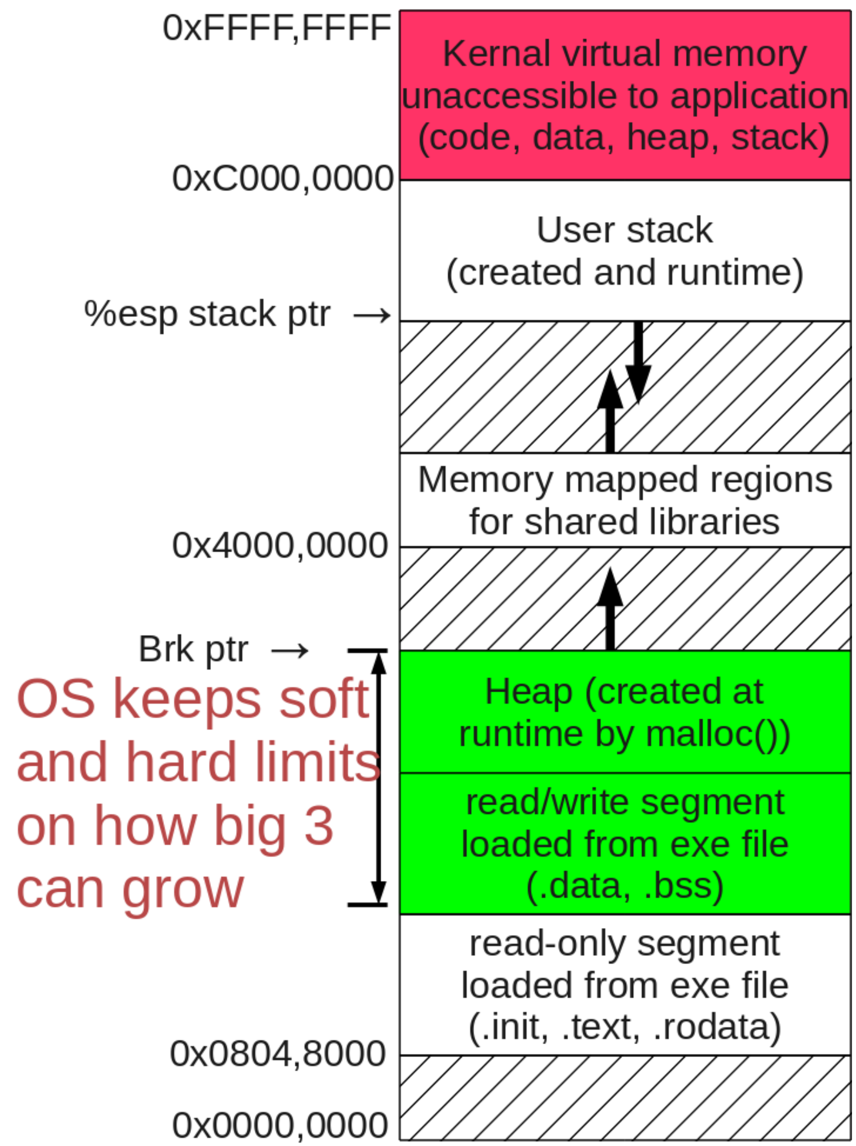
# OS's view of heap

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/resource.h>
#include <errno.h>
int     main    ()
{
  struct rlimit        resouceLimit;
  if  ( getrlimit(RLIMIT_DATA, &resouceLimit) )  {
    perror("getrlimit(RLIMIT_DATA) failed");
    return(EXIT_FAILURE);
  }

  printf("Process' combined max Data,"
        " ROData and heap sizes:\n");

  printf("Soft:\t");
  if  (resouceLimit.rlim_cur == RLIM_INFINITY)
    puts("(Unlimited)");
  else
    printf("%lu\n",resouceLimit.rlim_cur);

  printf("Hard:\t");
  if  (resouceLimit.rlim_max == RLIM_INFINITY)
    puts("(Unlimited)");
  else
    printf("%lu\n",resouceLimit.rlim_max);
  return(EXIT_SUCCESS);
}
```



# OS's view of heap, cont'd

- ***#include <unistd.h>***
- ***int brk(void *endDataSeg);***
  - sets end of data segment when
    - (1) value is reasonable,
    - (2) system has memory,
    - (3) process doesn't exceed max data size
- ***void *sbrk(intptr_t inc);***
  - increments data space by ***inc*** bytes
  - Not a system call, it is just a C library wrapper.
  - sbrk(0) finds current ptr value.
- ***<u>WARNING</u>***: YOU ARE ***malloc()*** and ***free()***!
  - Manually do what GNU C Library (glibc) does for you

# Check this out!

- A program that recursively
  1) Prints the current value of the `brk` pointer.
  2) Asks (in hexadecimal) how many bytes to allocate
  3) Allocates those bytes
  4) Prints the difference between the `brk` pointer and the end of the allocated block
  5) Recursively goes back to (1)

- Useful stuff:
  - `strtol("FF",NULL,16)` returns integer `0xFF`

# moveBrkPtr3.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

#define TEXT_LEN        10

void    doIt    ()
{
  char  text[TEXT_LEN];

  printf("sbrk is now                 "
         "                  %010p\n",
         sbrk(0)
        );
  printf("How much mem IN HEXADECIMAL "
         "do you want (0-8000,0=quit)? "
        );
  fgets(text,TEXT_LEN,stdin);
  int   size    = strtol(text,NULL,16);

  if  (size <= 0)
    return;

  void* ptr     = malloc(size);

  if  (errno == ENOMEM)
  {
    printf("Out of memory, Boss!\n");
    return;
  }

  printf("You just got addresses:"
         "       %010p - %010p.\n",
         ptr,ptr+(size-1)
        );
  printf("sbrk- blockEnd: %010p -"
         " %010p = %010p\n",
         sbrk(0),
         ptr+(size-1),
         sbrk(0)-(ptr+(size-1))
        );
```

# moveBrkPtr3.c, cont'd

```
  doIt();
  printf("Now freeing %p\n",ptr);
  free(ptr);
}


int     main    ()
{
  doIt();
  return(EXIT_SUCCESS);
}
```

# Using moveBrkPtr2.c

```
$ ./moveBrkPtr2
sbrk is now                                 0x0957e000
How much mem IN HEXADECIMAL do you want (0-8000, 0=quit)? 8000
You just got addresses:      0x0957e008 - 0x09586007.
sbrk- blockEnd: 0x095a7000 - 0x09586007 = 0x00020ff9
sbrk is now                                 0x095a7000 <= 1st malloc
How much mem IN HEXADECIMAL do you want (0-8000, 0=quit)? 8000
You just got addresses:      0x09586010 - 0x0958e00f.
sbrk- blockEnd: 0x095a7000 - 0x0958e00f = 0x00018ff1
sbrk is now                                 0x095a7000
How much mem IN HEXADECIMAL do you want (0-8000, 0=quit)? 8000
You just got addresses:      0x0958e018 - 0x09596017.
sbrk- blockEnd: 0x095a7000 - 0x09596017 = 0x00010fe9
sbrk is now                                 0x095a7000
```

# Using moveBrkPtr2.c

```
You just got addresses:      0x0958e018 - 0x09596017.
sbrk- blockEnd: 0x095a7000 - 0x09596017 = 0x00010fe9
sbrk is now                               0x095a7000
How much mem IN HEXADECIMAL do you want (0-8000, 0=quit)? 8000
You just got addresses:      0x09596020 - 0x0959e01f.
sbrk- blockEnd: 0x095a7000 - 0x0959e01f = 0x00008fe1
sbrk is now                               0x095a7000
How much mem IN HEXADECIMAL do you want (0-8000, 0=quit)? 8000
You just got addresses:      0x0959e028 - 0x095a6027.
sbrk- blockEnd: 0x095a7000 - 0x095a6027 = 0x00000fd9 <= too small
sbrk is now                               0x095a7000
How much mem IN HEXADECIMAL do you want (0-8000, 0=quit)? 8000
You just got addresses:      0x095a6030 - 0x095ae02f.
sbrk- blockEnd: 0x095cf000 - 0x095ae02f = 0x00020fd1
sbrk is now                               0x095cf000 <=sbrk changed
How much mem IN HEXADECIMAL do you want (0-8000, 0=quit)? 0
```

# Buffer overflow attacks

Buffer overflow:

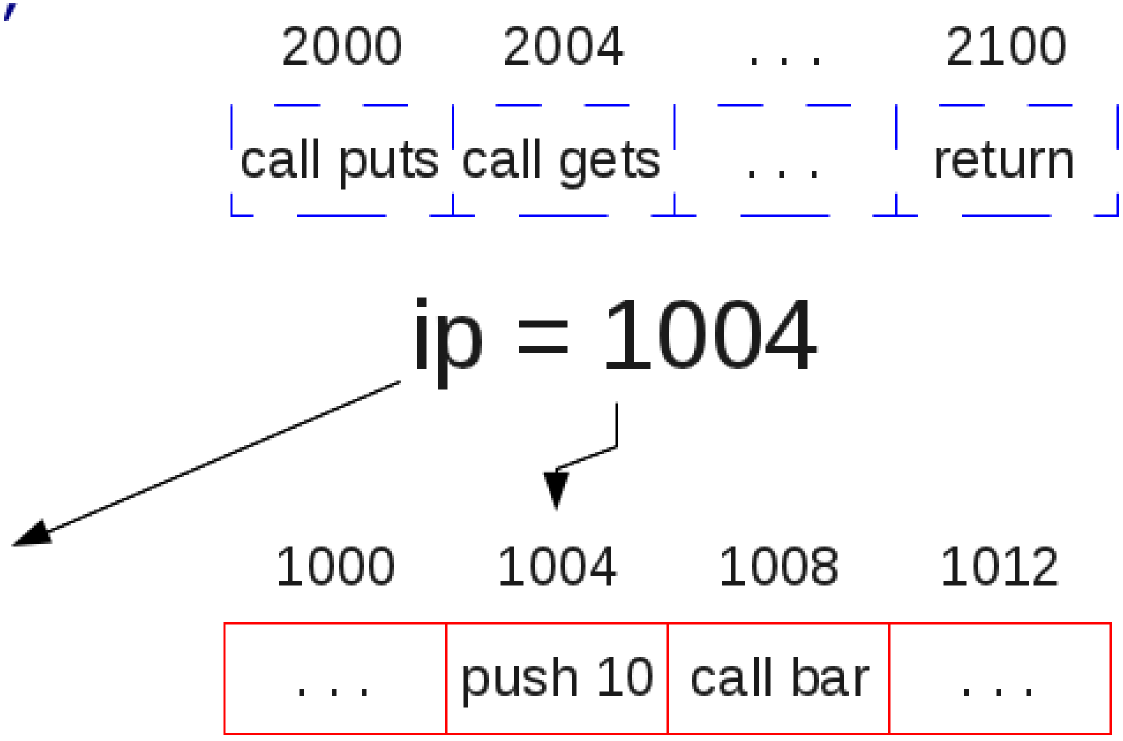
- Very common type of attack
- Exploits weakness in design of C with respect to:
  - implementation of arrays
  - Function calls
  - How values are saved in system stack
- So we'll have to review a wee bit of how function calls work . . .

# Buffer overflow attack (1)

```
void bar(int i)
{
   char buf[100];
   puts("Name?");
   gets(buf);
   . . .
   return;
}
```

```
void foo()
{
   . . .
   bar(10);
   . . .
}
```

Each function is in its own strip of mem. Instruction pointer (`ip`) points to current instruction being done



# Buffer overflow attack (2)

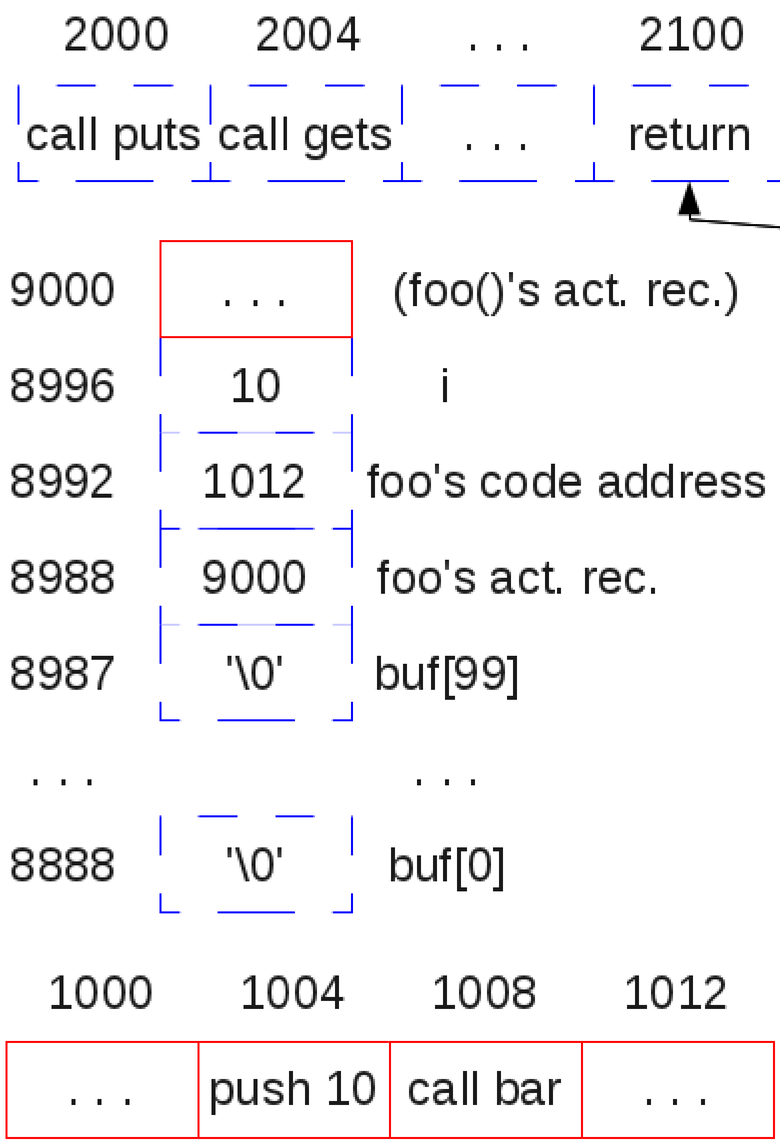
```
void bar(int i)
{
   char buf[100];
   puts("Name?");
   gets(buf);
   . . .
   return;
}

void foo()
{
   . . .
   bar(10);
   . . .
}
```

Local vars for each fnc are in an activation record on a sys stack, along with address of where to return in fnc that called you.

| | | |
|---|---|---|
| 9000 | . . . | (foo()'s act. rec.) |
| 8996 | 10 | i |
| 8992 | 1012 | foo's code address |
| 8988 | 9000 | foo's act. rec. |
| 8987 | '\0' | buf[99] |
| . . . | | . . . |
| 8888 | '\0' | buf[0] |

# Buffer overflow attack (3)



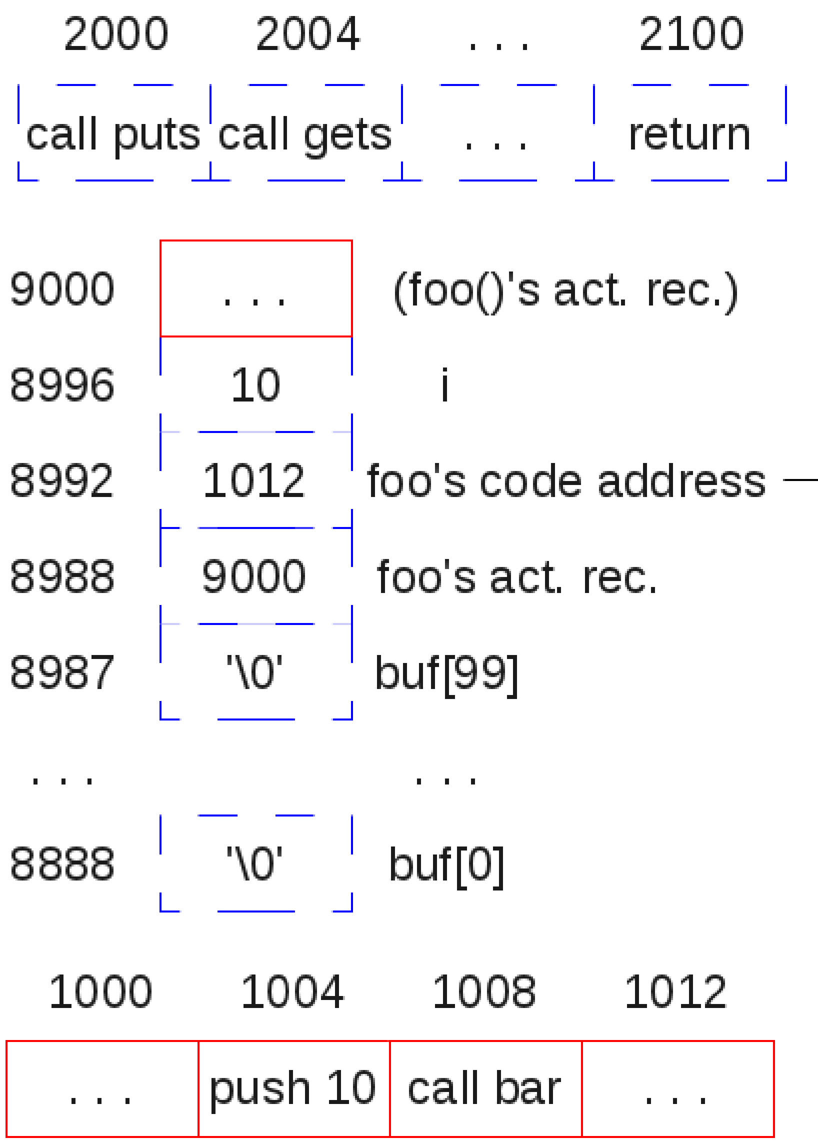

ip = 2100

(1) Instruction pointer gets to `bar()`'s `return`.

```
void bar(int i)
{
   char buf[100];
   puts("Name?");
   gets(buf);
   . . .
   return;
}

void foo()
{
   . . .
   bar(10);
   . . .
}
```

# Buffer overflow attack (4)

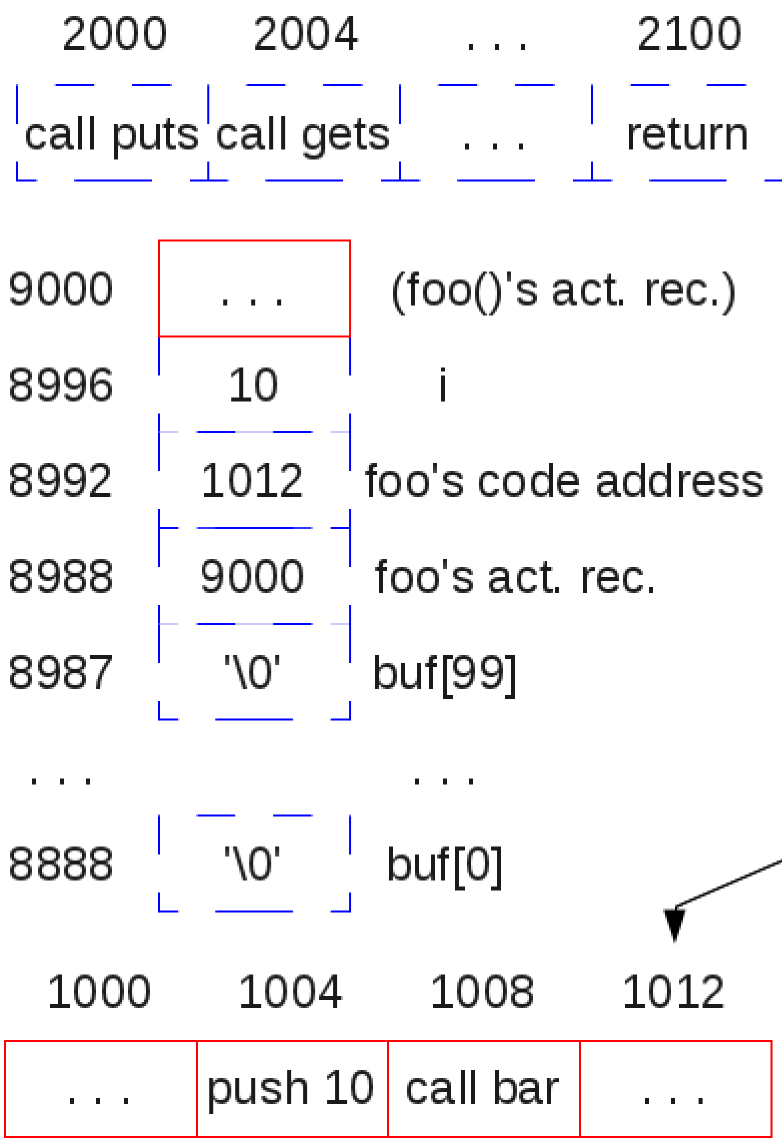



(2) Instruction pointer reads from `bar()`'s act. record address in `foo()` to `return` to.

```
void bar(int i)
{
   char buf[100];
   puts("Name?");
   gets(buf);
   . . .
   return;
}

void foo()
{
   . . .
   bar(10);
   . . .
}
```

# Buffer overflow attack (5)




(3) Program resumes where it left off in `foo()`.

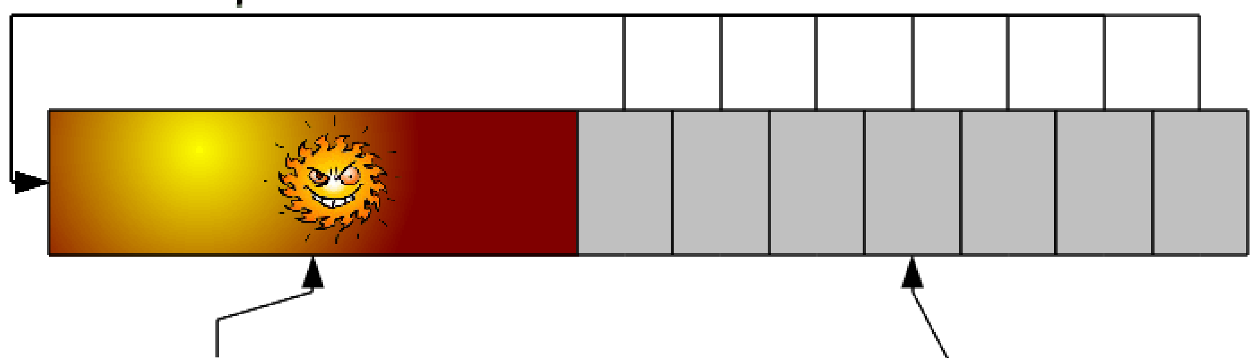
ip = 1012

```
void bar(int i)
{
   char buf[100];
   puts("Name?");
   gets(buf);
   . . .
   return;
}
```

```
void foo()
{
   . . .
   bar(10);
   . . .
}
```

# Buffer overflow attack (6)

(1) Malfoy writes a small bit of code to run a shell, and some extra code as pointers to beginning of the shell code.

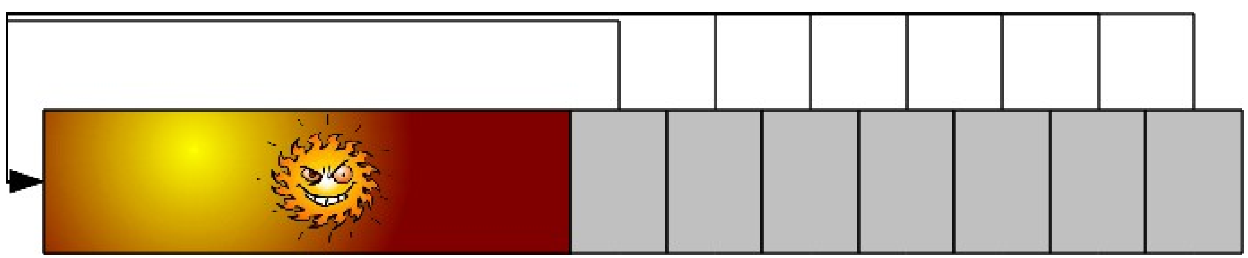


Code that does something bad! (like run a command line shell)

A bunch of pointers to the beginning of the bad part

# Buffer overflow attack (7)

(2) Malfoy gets machine code of the string



10 2E 4C B2 C4 07 08 . . .

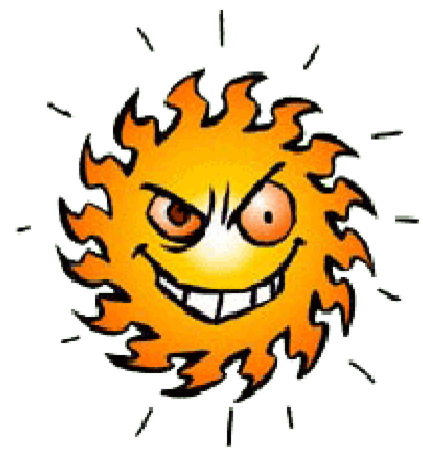
# Buffer overflow attack (8)

| 2000 | 2004 | . . . | 2100 |
|---|---|---|---|
| call puts | call gets | . . . | return |

ip = 2004

| | | |
|---|---|---|
| 9000 | . . . | (foo()'s act. rec.) |
| 8996 | 10 | i |
| 8992 | 1012 | foo's code address |
| 8988 | 9000 | foo's act. rec. |
| 8987 | '\0' | buf[99] |
| . . . | | . . . |
| 8888 | '\0' | buf[0] |

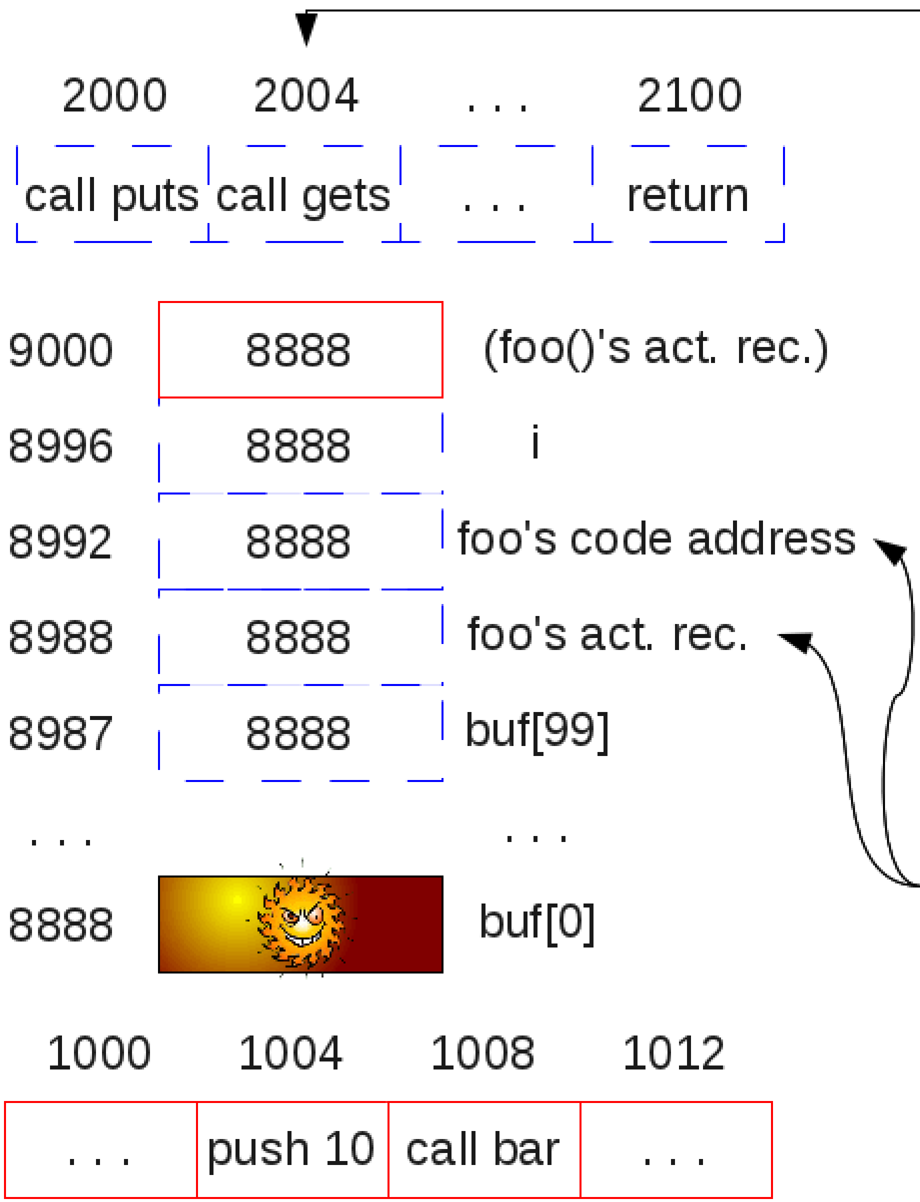
| 1000 | 1004 | 1008 | 1012 |
|---|---|---|---|
| . . . | push 10 | call bar | . . . |



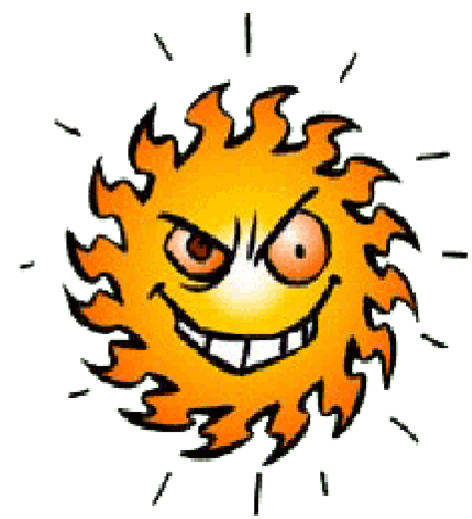
***"10 2E 4C B2 C4 07 08 . . ."***

(3) Malfoy inputs string as input to "Name?" request.

# Buffer overflow attack (9)




ip = 2004



(4) Input is so long it overwrites `buf[]` and info on `foo()`

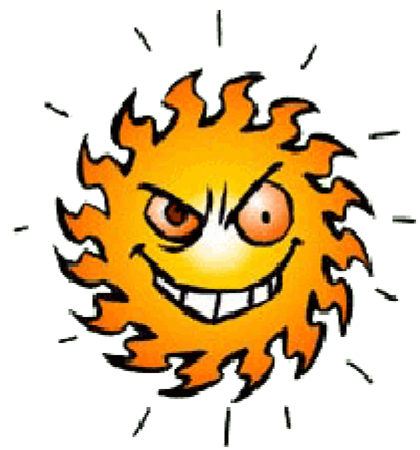
# Buffer overflow attack (10)

| 2000 | 2004 | . . . | 2100 |
|---|---|---|---|
| call puts | call gets | . . . | return |

ip = 2100

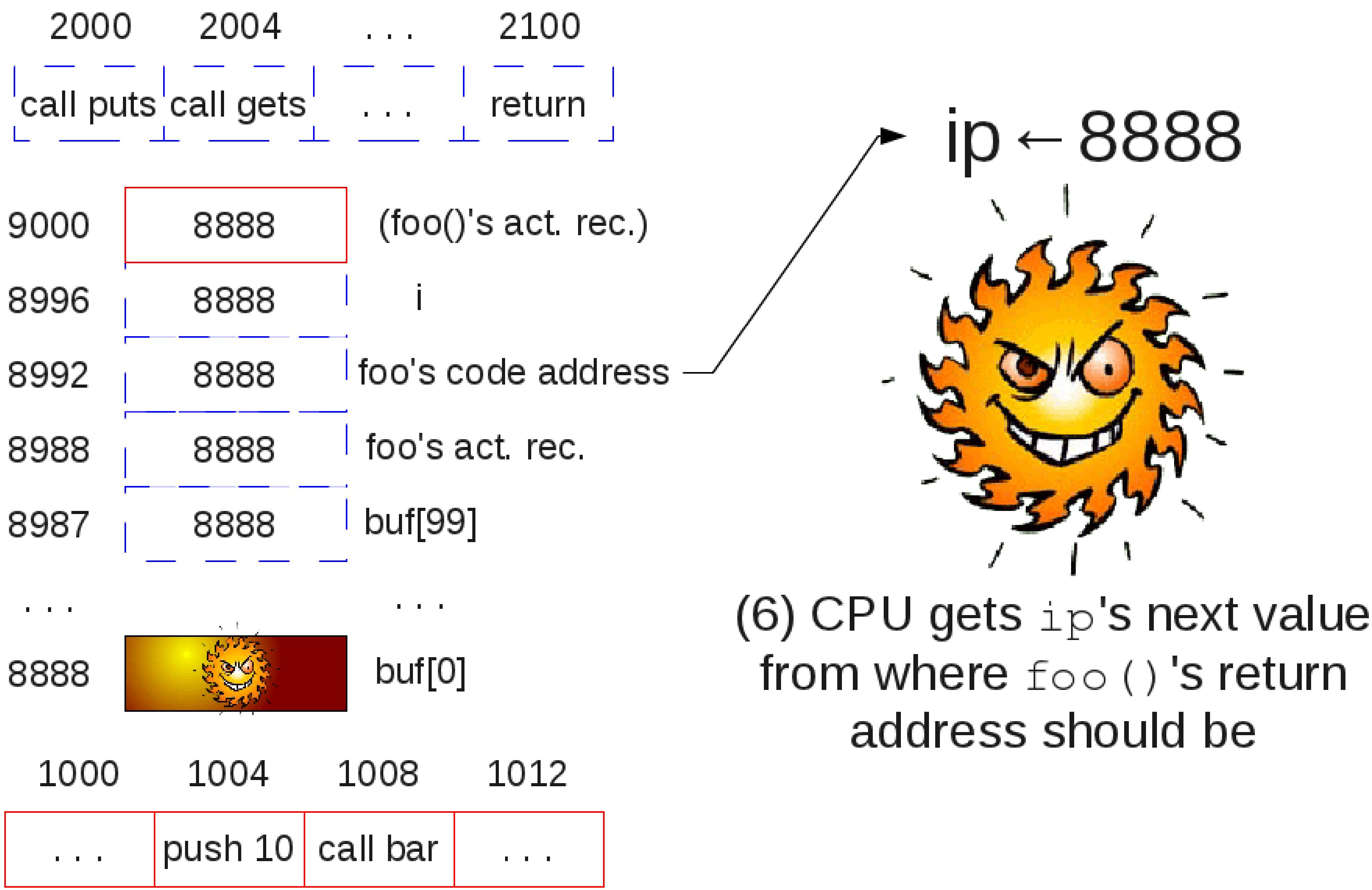
| | | |
|---|---|---|
| 9000 | 8888 | (foo()'s act. rec.) |
| 8996 | 8888 | i |
| 8992 | 8888 | foo's code address |
| 8988 | 8888 | foo's act. rec. |
| 8987 | 8888 | buf[99] |
| . . . | | . . . |
| 8888 |  | buf[0] |

| 1000 | 1004 | 1008 | 1012 |
|---|---|---|---|
| . . . | push 10 | call bar | . . . |



(5) Function `bar()` finishes

# Buffer overflow attack (11)



# Buffer overflow attack (12)

| 2000 | 2004 | . . . | 2100 |
|---|---|---|---|
| call puts | call gets | . . . | return |

| | | |
|---|---|---|
| 9000 | 8888 | (foo()'s act. rec.) |
| 8996 | 8888 | i |
| 8992 | 8888 | foo's code address |
| 8988 | 8888 | foo's act. rec. |
| 8987 | 8888 | buf[99] |
| . . . | | . . . |
| 8888 | | buf[0] |

| 1000 | 1004 | 1008 | 1012 |
|---|---|---|---|
| . . . | push 10 | call bar | . . . |


ip = 8888

(7) Now system is running Malfoy's nasty code!

# Preventing Buffer Overflow

- Solution #1: Use C++ string objects.
  - `const char* string::c_str()`;
  - Returns C-string representation of C++ string
- Solution #2: Use safe C-string fncs

```
Bad:  gets(char* buf)
Good: fgets(char* buf,size_t size,FILE* filePtr)

Bad:  sprintf(char* buf, ...)
Good: snprintf(char* buf,size_t size, ...)

Bad:  strcpy(char* to,char* from)
Good: strncpy(char* to,char* from,size_t size)

Bad:  strcat(char* to,char* from)
Good: strncat(char* to,char* from,size_t size)

Bad:  strcmp(char* p0,char* p1)
Good: strncmp(char* p0,char* p1,size_t size)
```

# Preventing Buffer Overflow, cont'd

- **`strncpy(char* to, const char* from, size_t size)`** will copy **`size`** bytes <u>*without*</u> copying **`'\0'`** if no **`'\0'`** is present in **`from`**.
- **`strncat(char* to, const char* from, size_t size)`** will copy up to **`size`** bytes from **`from`** and will <u>*always*</u> copy or add **`'\0'`**. Thus, it could write up to **`size+1`** bytes.

# Your turn! Fix this program!

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define STRING_LEN 20 /*Assumed char array len */
#define NUMBER_LEN 3

void enterName(char*  nPtr)
{ printf("Your name: ");
  gets(nPtr);
}

void enterAge(int* agePtr)
{ char numberText[NUMBER_LEN];
  printf("Your age: ");
  gets(numberText);
  *agePtr=strtol(numberText,0,0);
}
```

# Your turn! Fix this (cont'd)

```
void enterFavoriteColor  (const char* itemNamePtr,
                                 char*    entryPtr)
{ printf("Fav. color for %s.",itemNamePtr);
  gets(entryPtr);
}

void printInfo (char*  nameP, int    y,
                char* carCP,char* houseCP)
{
 char   text[STRING_LEN];
 sprintf(text,"%s who's %d yrs old",nameP,y);
 printf ("%s likes car color %s",text,carCP);
 if(strcmp(carCP,houseCP) == 0)
  puts("They like same color for houses");
 else
  printf("They like house color %s",houseCP);
}
```

# Your turn! Fix this (cont'd)

```
int  main ()
{
  char name[STRING_LEN];
  int  age;
  char carColor[STRING_LEN];
  char houseColor[STRING_LEN];

  enterName(name);
  enterAge(&age);
  enterFavoriteColor("car",carColor);
  enterFavoriteColor("house",houseColor);
  printInfo(name,age,carColor,houseColor);
  return(EXIT_SUCCESS);
}
```

# Your turn again!

- Revise the **`naughtyCopy()`** to copy at most **`size_t n`** chars.

```
char* naughtyCopy(const char* fromP, size_t
n)
{
  char* toP;

  for  ( ;  *fromP != '\0'; fromP++, toP++)
    *toP = *fromP;

  free(fromP);
  return(toP);
}
```

Next time:
Input/Output!