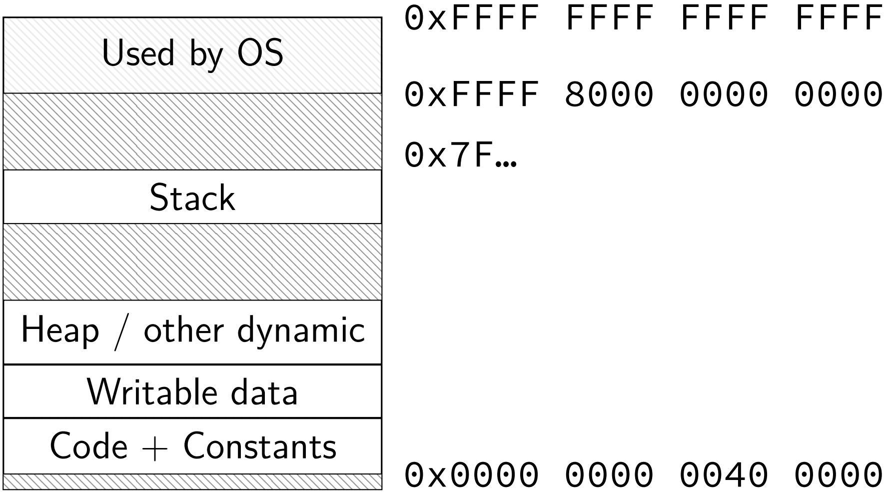
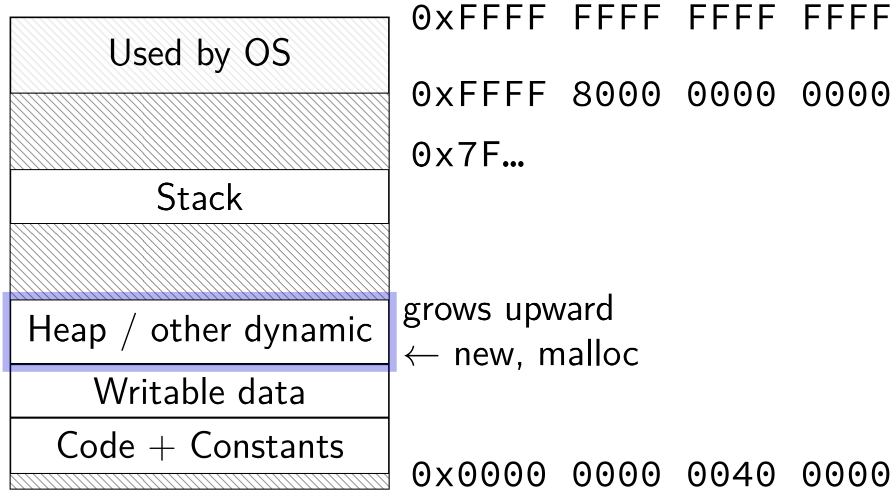
# memory

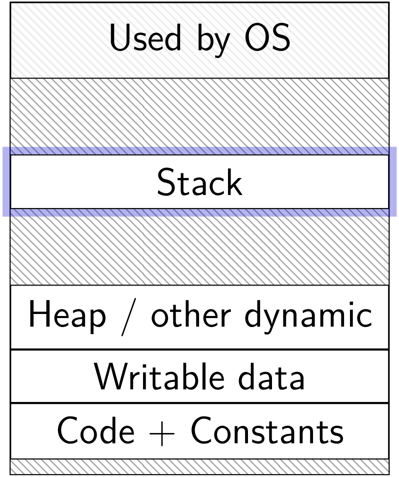
# a possible memory layout on Linux



# a possible memory layout on Linux



# a possible memory layout on Linux




0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

← local variables allocated here

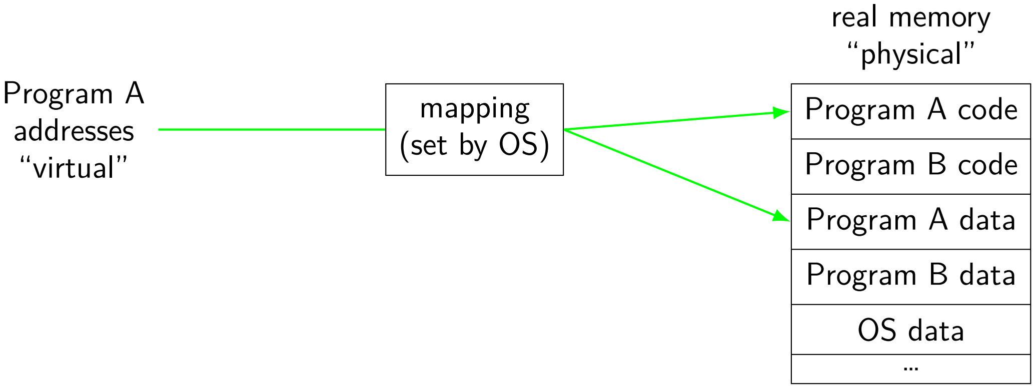
grows downward

0x0000 0000 0040 0000

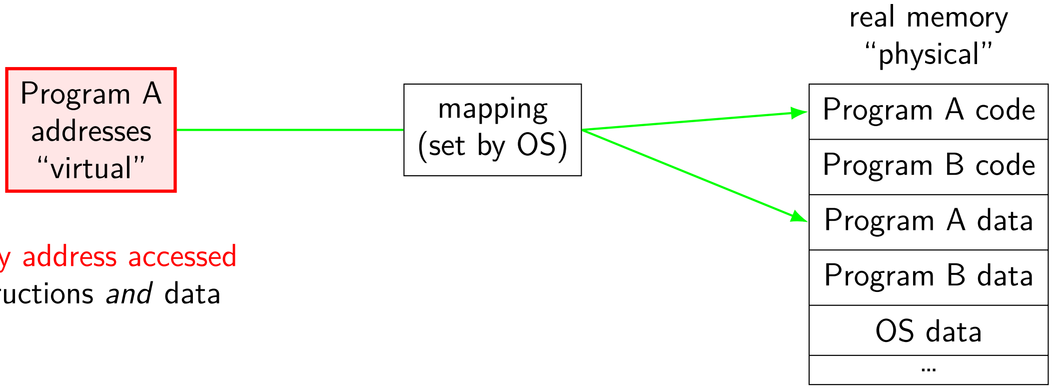
# stack v heap

| stack | heap |
|---|---|
| compiler managed | programmer managed |
| values go out of scope | explicit free |
| within procedure only | outlives procedures |
| x86: grows down | x86: grows up |

# address translation

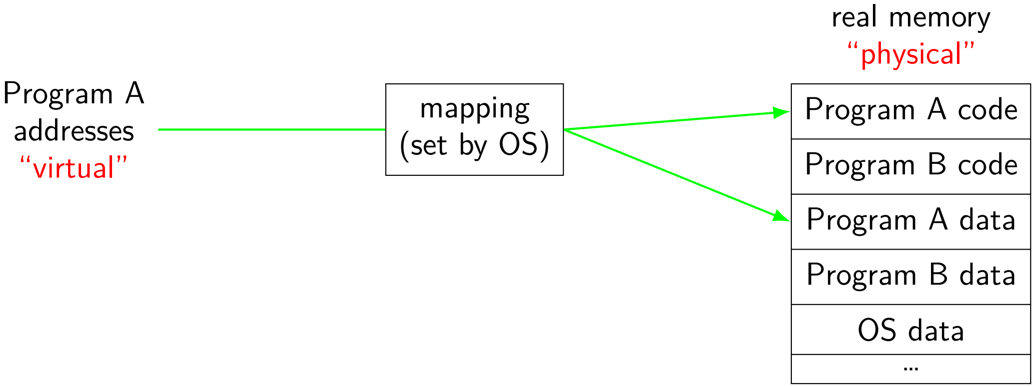


# address translation



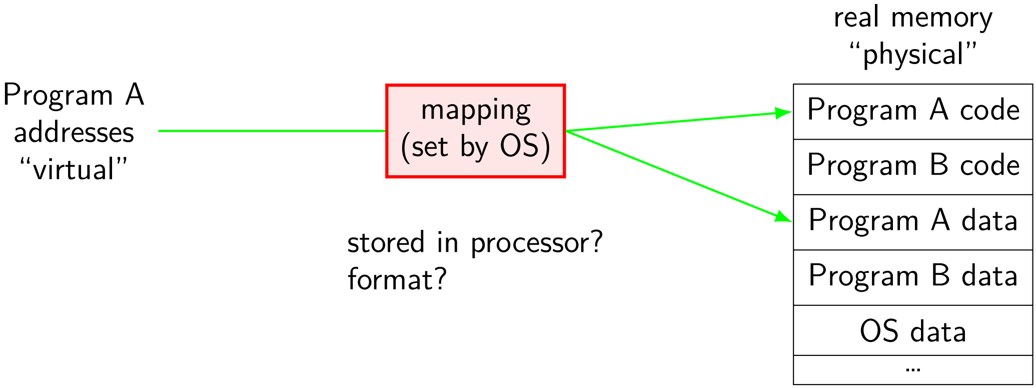

every address accessed
instructions *and* data

# address translation




program addresses are 'virtual'
real addresses are 'physical'

# address translation



# aside: void *

generic pointer type

cannot dereference!

in C: no casts needed

in C++: casts needed

# aside: size_t

unsigned integer type

big enough to hold size of anything allocated

x86-64: typically same as `unsigned long`

# alloca

```
ALLOCA(3)                  Linux Programmer's Manual                 ALLOCA(3)

NAME
       alloca - allocate memory that is automatically freed

SYNOPSIS
       #include <alloca.h>

       void *alloca(size_t size);

DESCRIPTION
       The  alloca()  function allocates size bytes of space in the stack frame of the caller.
       This temporary space is automatically freed when  the  function  that  called  alloca()
       returns to its caller.
```

# writing alloca

how is it possible to write this function???

allocating space without overwriting return address???

# an historical implementation

386BSD (1990) 32-bit x86 implementation

converted to Intel syntax, some comments added

```
alloca:
    pop edx              /*  pop return addr */
    pop eax              /*  pop amount to allocate */
    mov ecx, esp
    add eax, 3           /*  round up to next word */
    and eax, 0xfffffffc
    sub esp,eax          /* adjust stack pointer for allocation */
    mov eax,esp          /* set ret. val. to base of
                                newly allocated space */
    push [ecx+8]         /* copy possible saved registers */
    push [ecx+4]
    push [ecx+0]
    push eax             /* dummy to pop at callsite */
    jmp edx              /* "return" */
```

# an historical implementation

386BSD (1990) 32-bit x86 implementation

converted to Intel syntax, some comments added

```
alloca:
    pop edx                 /*  pop return addr */
    pop eax                 /*  pop amount to allocate */
    mov ecx, esp
    add eax, 3              /*  round up to next word */
    and eax, 0xfffffffc
    sub esp,eax             /* adjust stack pointer for allocation */
    mov eax,esp             /* set ret. val. to base of
                                   newly allocated space */
    push [ecx+8]            /* copy possible saved registers */
    push [e
    push [ecx+0]
    push eax                /* dummy to pop at callsite */
    jmp edx                 /* "return" */
```

32-bit x86 calling convention: all args on stack

# an historical implementation

386BSD (1990) 32-bit x86 implementation

converted to Intel syntax, some comments added

changing stack pointer

how does caller access local variables on the stack?

assumption: uses a base pointer instead...

```
alloca:
    pop ec
    pop ea
    mov ec
    add eax, 3              /* round up to next word */
    and eax, 0xfffffffc
    sub esp,eax             /* adjust stack pointer for allocation */
    mov eax,esp             /* set ret. val. to base of
                                   newly allocated space */
    push [ecx+8]            /* copy possible saved registers */
    push [ecx+4]
    push [ecx+0]
    push eax                /* dummy to pop at callsite */
    jmp edx                 /* "return" */
```

# an historical implementation

386BSD (1990) 32-bit x86 implementation

converted to Intel syntax, some comments added

how do they know caller only saves 3 registers?

maybe they wrote the compiler...?

```
alloca:
    pop edx
    pop eax
    mov ecx
    add eax, 3              /* round up to next word */
    and eax, 0xfffffffc
    sub esp,eax             /* adjust stack pointer for allocation */
    mov eax,esp             /* set ret. val. to base of
                                   newly allocated space */
    push [ecx+8]            /* copy possible saved registers */
    push [ecx+4]
    push [ecx+0]
    push eax                /* dummy to pop at callsite */
    jmp edx                 /* "return" */
```

# a modern implementation: compiler built-in

```c
void foo(int N) {
    char *temp = alloca(N);
    bar(temp);
}
```

```
foo: # @foo
  push rbp
  mov rbp, rsp
  movsxd rax, edi
  mov rdi, rsp
  add rax, 15
  and rax, -16
  sub rdi, rax
  mov rsp, rdi
  call bar
  mov rsp, rbp
  pop rbp
  ret
```

# a modern implementation: compiler built-in

```
void foo(int N) {
    char *temp = alloca(N);
    bar(temp);
}
```

```
foo: # @foo
  push rbp
  mov rbp, rsp
  movsxd rax, edi
  mov rdi, rsp
  add rax, 15
  and rax, −16
  sub rdi, rax
  mov rsp, rdi
  call bar
  mov rsp, rbp
  pop rbp
  ret
```

use frame pointer —
remember original stack location

# a modern implementation: compiler built-in

```
void foo(int N) {
    char *temp = alloca(N);
    bar(temp);
}
```

```
foo: # @foo
  push rbp
  mov rbp, rsp
  movsxd rax, edi
  mov rdi, rsp
  add rax, 15
  and rax, −16
  sub rdi, rax
  mov rsp, rdi
  call bar
  mov rsp, rbp
  pop rbp
  ret
```

rsp becomes rsp - N
(N rounded up to next mult. of 16)

# malloc

```
void *malloc(size_t size);
```

`size_t` — integer type that holds size (in bytes)

# typical malloc usage

```
int *array;
...
array = malloc(number_of_elements * sizeof(*array))
// OR
array = malloc(number_of_elements * sizeof(int))
```

---

```
SomeType *item;
...
item = malloc(sizeof(*item));
// OR
item = malloc(sizeof(SomeType));
```

---

note: in C++ (not C) would need casts

```
array = (int*) malloc(...);
```

# malloc and free

free — undo malloc's allocation

# new

new does two things that can be done seperately

allocate memory

```
operator new(sizeof(Foo))
```

call constructors

can do separately with “placement new”

```
new (somePtr) Foo(arguments);
```

# “manually” doing what new does

```
Foo *foo = new Foo(1, 2, 3);
```

---

```
#include <memory>  // prototypes for operator new
...

// allocate space
Foo *foo = (Foo*) operator new(sizeof(Foo));

// call constructor
new (foo) Foo(1, 2, 3);
```

# implementing vector: create

```
template <class T> class MyVector {
    ...
private:
    T * array;
    int size, capacity;
};

template <class T>
void MyVector::push_back(const T& other) {
    // increase array capacity if needed
    if (++size > capacity) { ... }

    // call copy constructor to create array[size-1]
    new (&array[size - 1]) T(other);
        // better than constructing all in advance and assigning
        // e.g. if vector of lists,
        //      don't allocate "extra" head/tail dummy nodes
}
```

# delete

delete does two things that can be done seperately

call destructors

```
foo->~Foo();
```

actually free memory

```
operator delete(foo);
```

# implementing vector: destroy

```
template <class T> class MyVector {
    ...
private:
    T * array;
    int size, capacity;
};

template <class T>
void MyVector::pop_back(const T& other) {
    size--;
    array[size].~T();
}
```

# implementing malloc

| malloc/new | OS allocation interfaces |
| --- | --- |
| 16 byte or smaller allocations | minimum allocation/free: 4KB |
| 100ish ns/allocation or free | microsecondish allocation/free |

OS manages memory in **4KB pages**

malloc/new "batch" small allocations into these big requests

# implementing malloc/free

get large allocations from OS

subdivide allocation — need data structure to manage

- one idea: before what malloc/new returns
- another idea: separate, e.g., hashtable on address

lots of tricky choices:

- what if there are lots of non-contiguous free chunks?
- how to quickly find chunk of appropraite size
- ...

# implementing malloc/free

get large allocations from OS

subdivide allocation — need data structure to manage

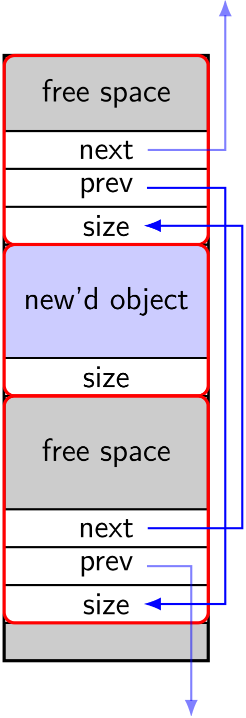
- one idea: before what malloc/new returns
- another idea: separate, e.g., hashtable on address

lots of tricky choices:

- what if there are lots of non-contiguous free chunks?
- how to quickly find chunk of appropraite size
- ...

# one malloc/free impl.

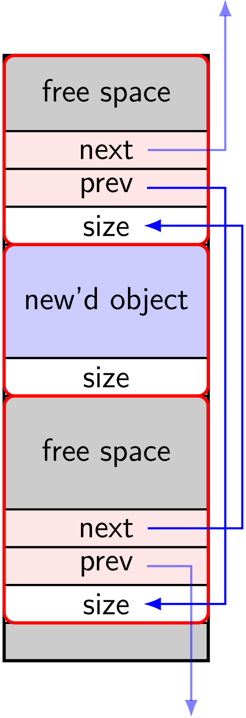
```
struct AllocInfo {
    int size;
    // for alloc'd:
    AllocInfo *prev;
    AllocInfo *next;
};
```



# one malloc/free impl.

```
struct AllocInfo {
  int size;
  // for alloc'd:
  AllocInfo *prev;
  AllocInfo *next;
};
```

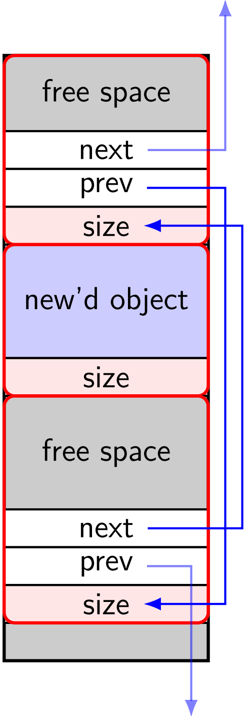
keep linked list of available chunks of memory



# one malloc/free impl.

```
struct AllocInfo {
  int size;
  // for alloc'd:
  AllocInfo *prev;
  AllocInfo *next;
};
```

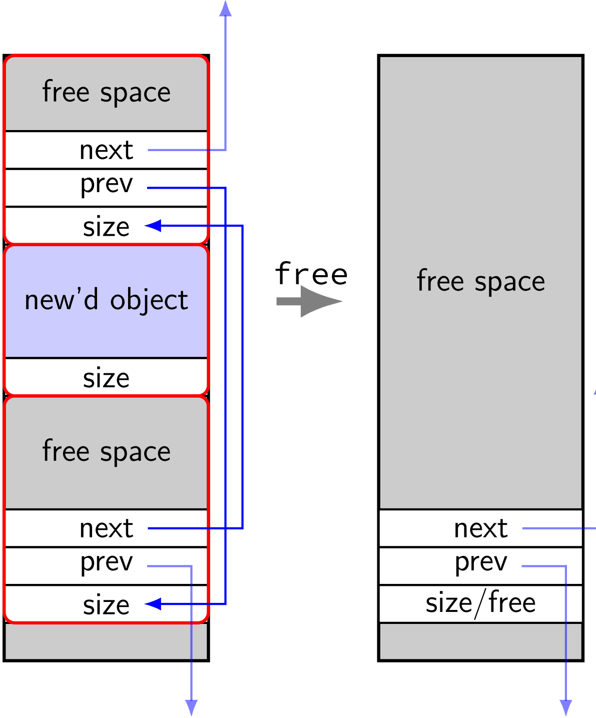
keep sizes before allocations
maybe need less with `delete`?



# one malloc/free impl.

```
struct AllocInfo {
    int size;
    // for alloc'd:
    AllocInfo *prev;
    AllocInfo *next;
};
```

merge adjacent free allocations (if any)



# tough malloc/free choices

quickly finding free blocks of right size

avoiding large amounts of small, free spaces

- enough free memory, but not usable?
- "fragmentation"

extra overhead (sizes, next/prev pointers, ...)

how many lists of free blocks?

different lists for different sizes?

return first block or best sizes block? in between?

# realloc

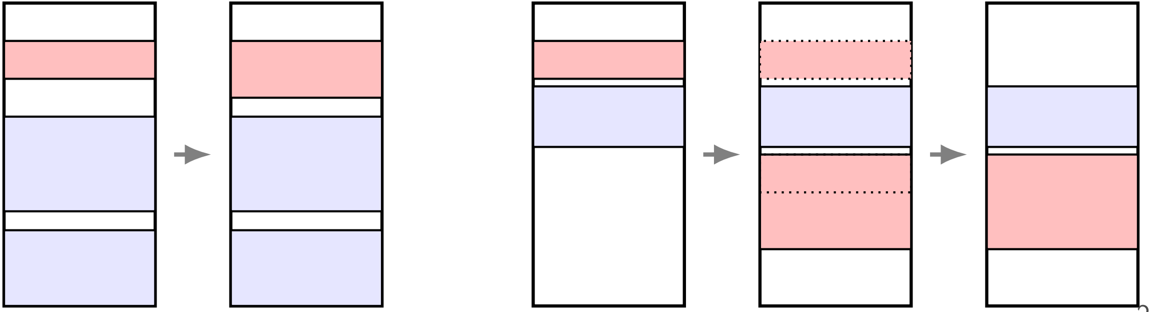
`void *realloc(void *pointer, size_t size)`

either:

changes the size of the allocation at `pointer`, or

allocates new space, copies data from `pointer` there, free (old) pointer

returns the new space (if any, or `pointer` otherwise)



# realloc

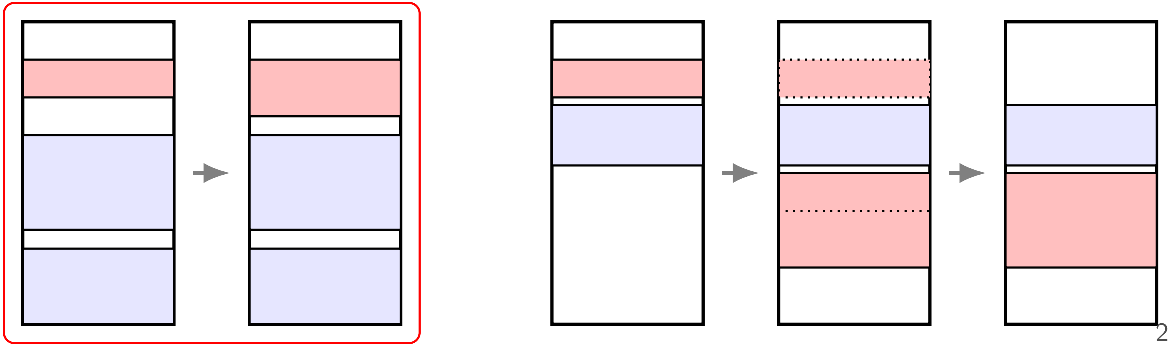
```
void *realloc(void *pointer, size_t size)
```

either:

changes the size of the allocation at `pointer`, or

allocates new space, copies data from `pointer` there, free (old) pointer

returns the new space (if any, or `pointer` otherwise)



# realloc

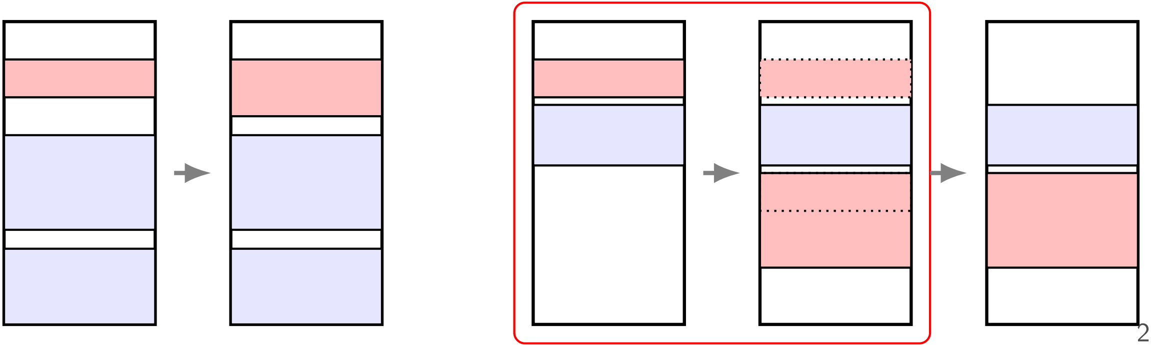
```
void *realloc(void *pointer, size_t size)
```

either:

changes the size of the allocation at `pointer`, or

allocates new space, copies data from `pointer` there, free (old) pointer

returns the new space (if any, or `pointer` otherwise)



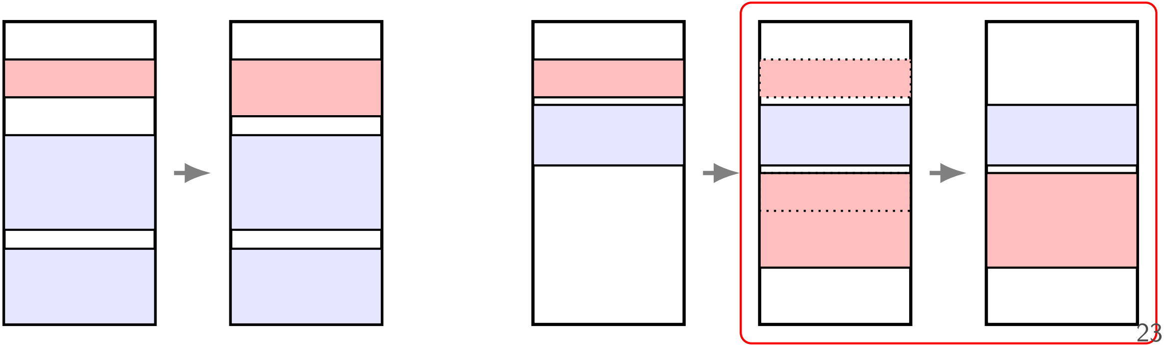
# realloc

`void *realloc(void *pointer, size_t size)`

either:

- changes the size of the allocation at `pointer`, or
- allocates new space, copies data from `pointer` there, free (old) pointer

returns the new space (if any, or `pointer` otherwise)



# some realloc gotchas

need to use return value — data might have moved!

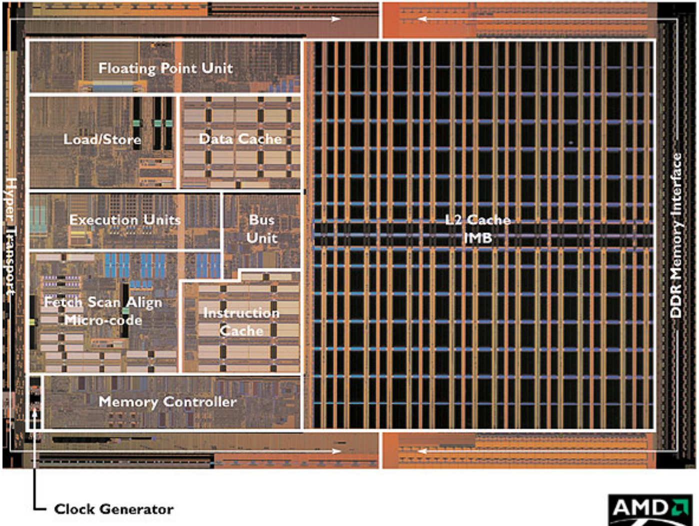
need to worry about other copies of the pointer

# realloc runtime

copy: $\Theta(n)$

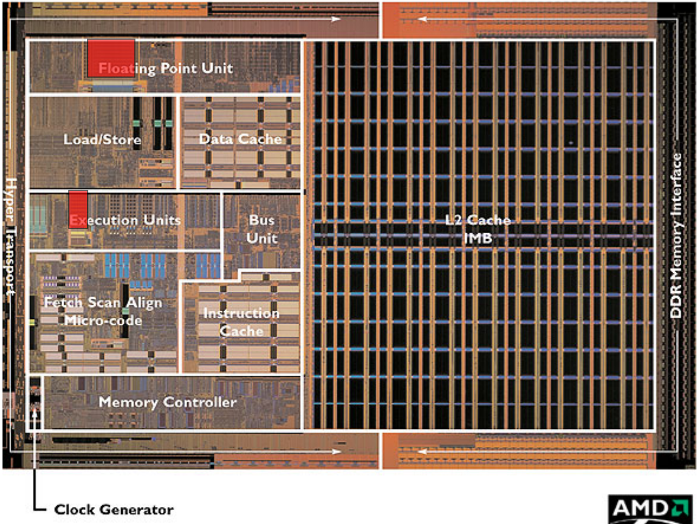
in place: $\Theta(1)$

# 2004 CPU




age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 2004 CPU




Registers

Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 2004 CPU

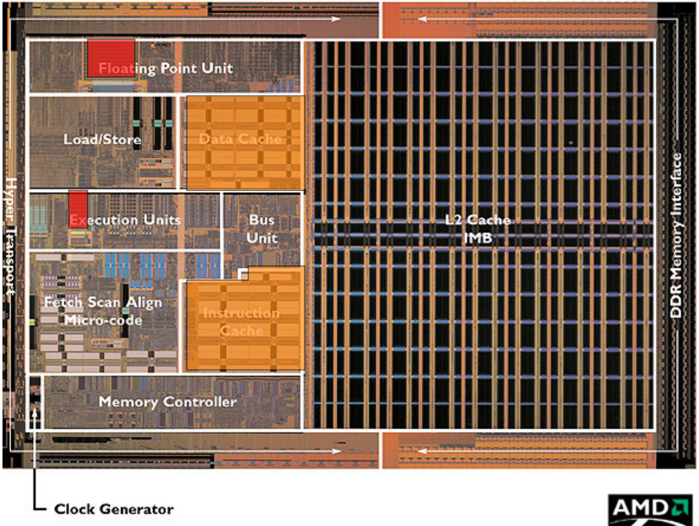
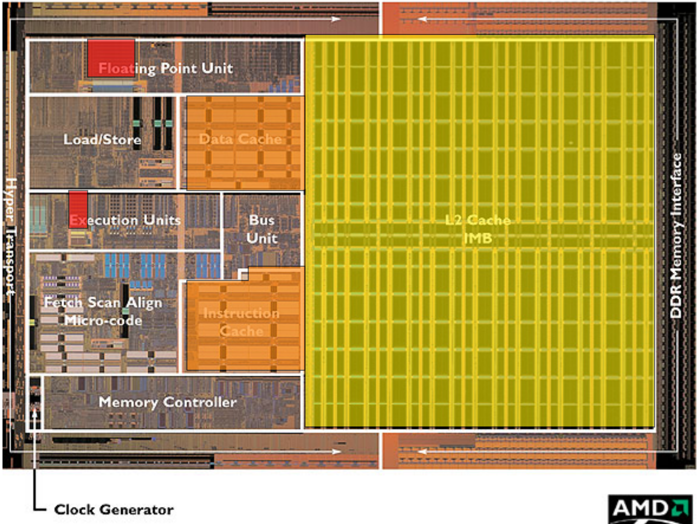
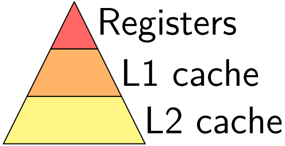




Image: approx 2004 AMD press image of Opteron die; approx register location via chip-architect.org (Hans de Vries)

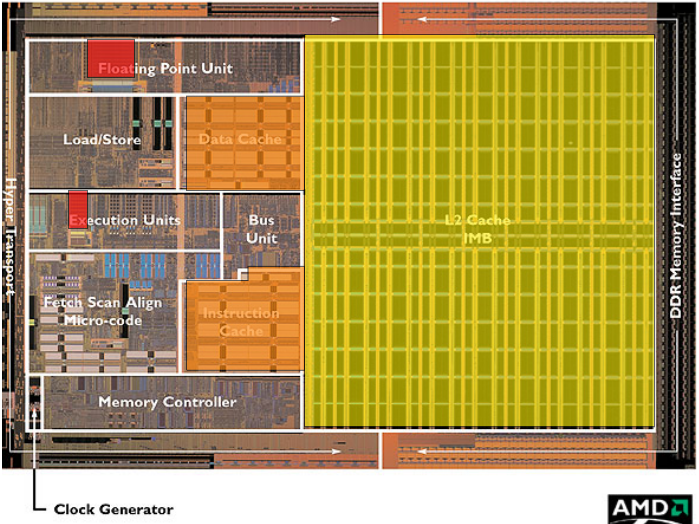
# 2004 CPU




age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 2004 CPU



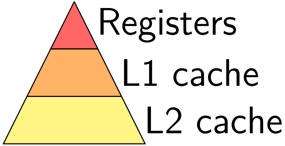
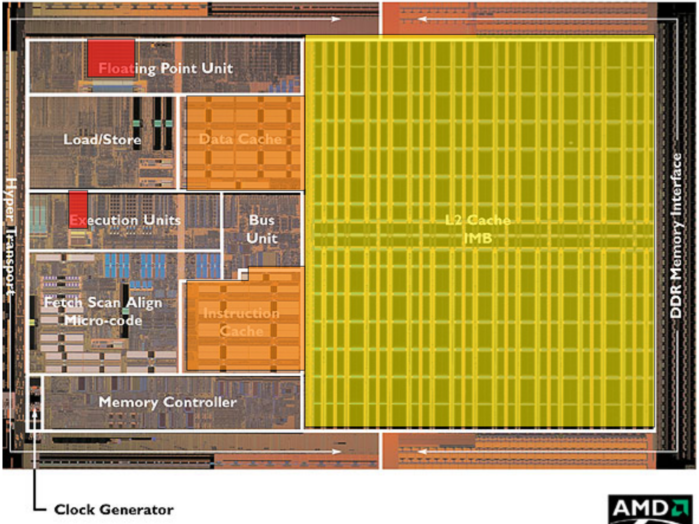
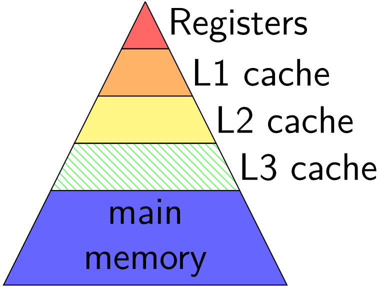




Image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

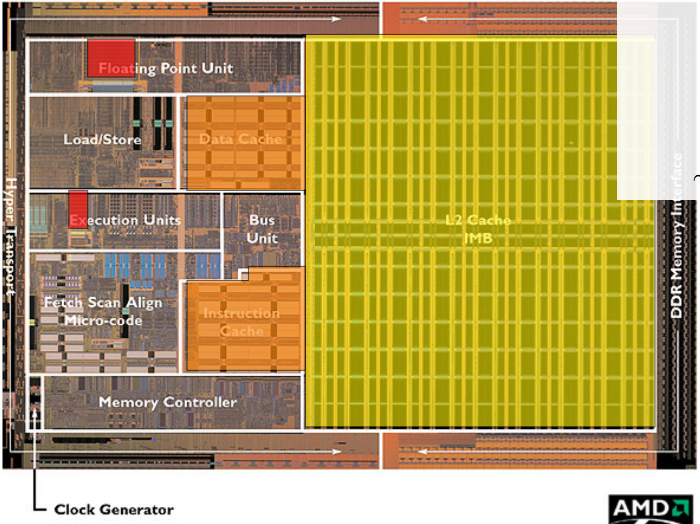
# 2004 CPU




age: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# 2004 CPU



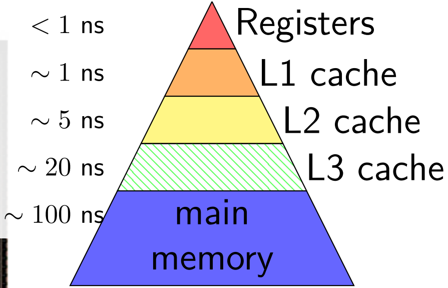
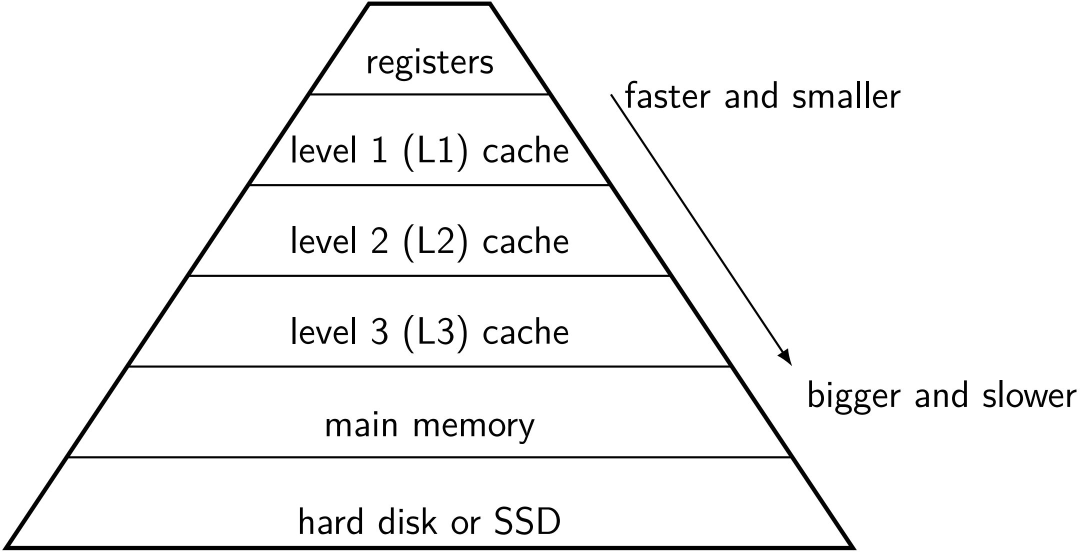




image: approx 2004 AMD press image of Opteron die;
approx register location via chip-architect.org (Hans de Vries)

# memory hierarchy overview



# memory hierarchy goal

size of largest, slowest storage

speed of smallest, fastest storage

not actually possible, but can get pretty close due to locality

# memory hierarchy numbers

from a system like my desktop:

(note: multiple parallel accesses and/or sequential accesses needed to achieve maximum bandwidths)

| level | time/access | maximum read bandwidth |
|---|---|---|
| registers | 0.3 ns | $\sim$ 645 GB/s (per core) |
| L1 cache | 1.2 ns | $\sim$ 199 GB/s (per core) |
| L2 cache | 3.6 ns | $\sim$ 110 GB/s (per core) |
| L3 cache | $\sim$ 13 ns | $\sim$ 54 GB/s |
| main memory | $\sim$ 64 ns | $\sim$ 25 GB/s |
| hard disk | $\sim$ 5 000 000 ns | $\sim$ 0.1 GB/s |

# caches

caches — fast memory that holds
recently accessed values from main memory and
values near recently accessed values from main memory

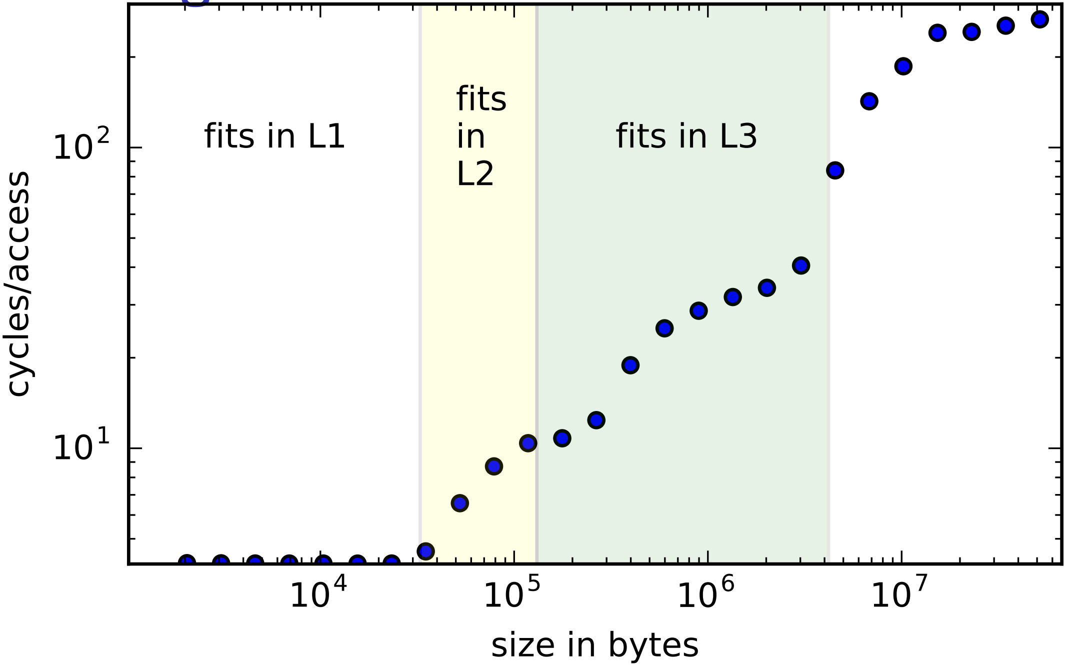
idea: program thinks it accesses main memory…
but most accesses take 'shortcut' to cache

# observing caches

```
unsigned run(int count) {
    unsigned index = 1;
    for (unsigned j = 0; j < count; ++j) {
        // use array @ index to find next index
        // prevents parallel accesses to cache/memory
        index = array[index];
    }
    return index;
}

// setup to access array with bad spatial locality
// size is the approx. # elements to access
void setup(int size) {
    for (int i = 0; i < size; ++i)
        order[i] = i;
    randomlyShuffle(order, size);
    for (int i = 0; i < size - 1; ++i) {
        /* order[i] should point to order[i+1] */
        array[order[i]] = order[(i + 1) % size];
```

# observing caches



# memory hierarchy assumptions

temporal locality
“if a value is accessed now, it will be accessed again soon”

caches should keep recently accessed values

spatial locality
“if a value is accessed now, adjacent values will be accessed soon”

caches should store adjacent values at the same time

natural properties of programs — think about loops

# locality examples

```
double computeMean(int length, double *values) {
    double total = 0.0;
    for (int i = 0; i < length; ++i) {
        total += values[i];
    }
    return total / length;
}
```

temporal locality: machine code of the loop

spatial locality: machine code of most consecutive instructions

temporal locality: `total`, `i`, `length` accessed repeatedly

spatial locality: `values[i+1]` accessed after `values[i]`

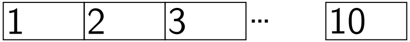
# locality example

```
for ( int i = 0; i < 1024; i++ )
    for ( int j = 0; j < 1024; j++ )
        array[i][j] = 0;
for ( int c = 0; c < 1024; c++ )
    for ( int i = 0; i < 1024; i++ )
        for ( int j = 0; j < 1024; j++ )
            array[i][j]++;
for ( int i = 0; i < 1024; i++ )
    for ( int j = 0; j < 1024; j++ )
        sum += array[i][j];
```

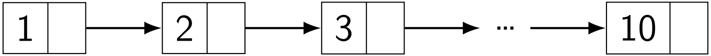
on my laptop: 0.30 s

```
for ( in
    for
for ( in
    for
for ( in
    for
```

# data structure locality



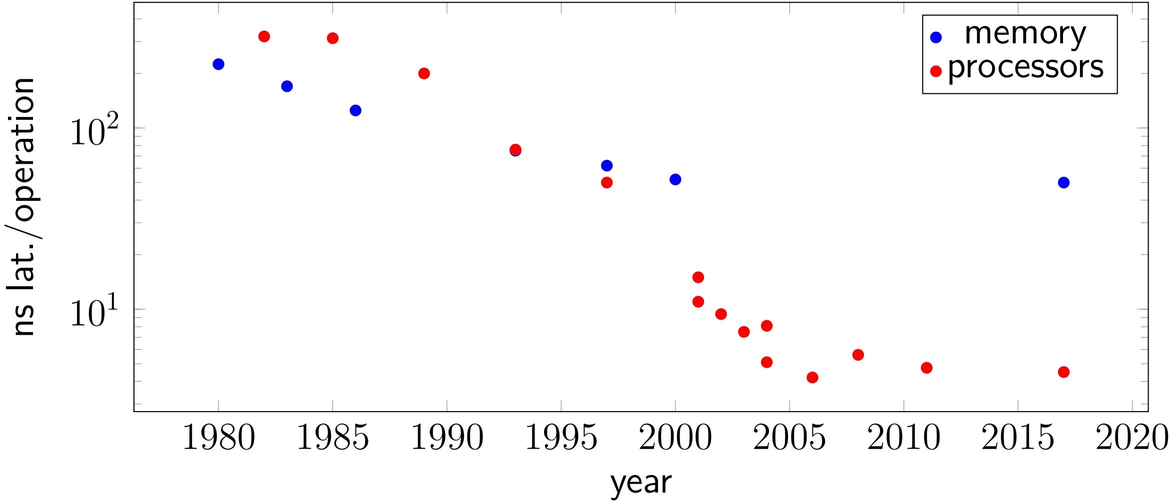

| | |
|---|---|
| 0x1000 | 1 |
| 0x1008 | 2 |
| 0x1010 | 3 |
| 0x1018 | 4 |
| … | … |




| | |
|---|---|
| 0x1000 | 1 |
| 0x1008 | 0x1050 |
| … | … |
| 0x1020 | 3 |
| 0x1028 | 0x1060 |
| … | … |
| 0x1050 | 2 |
| 0x1058 | 0x1020 |
| 0x1060 | 4 |
| … | … |

# CPU/memory time per operation

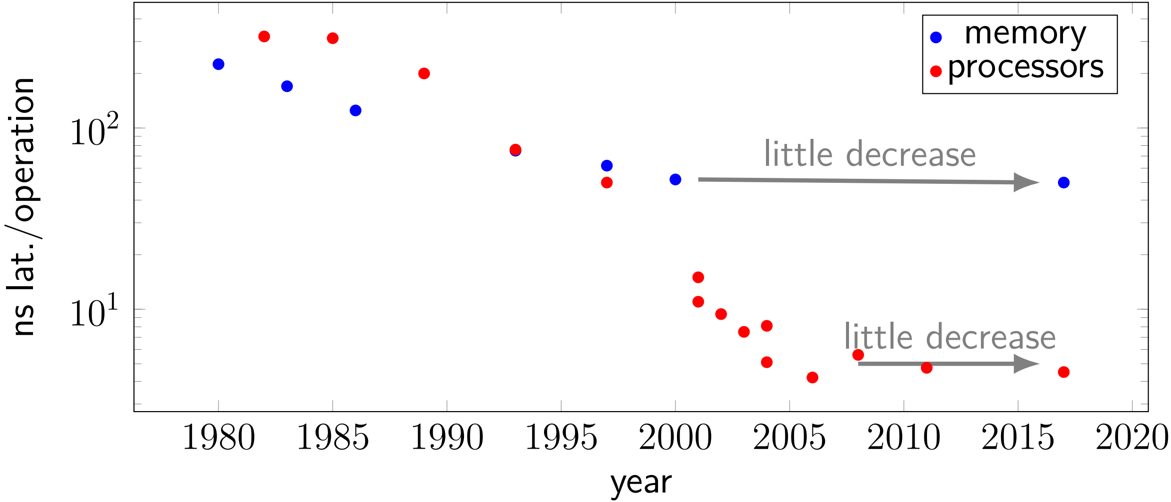
(everything approximate...)




data up to 2001, data via Patterson, "Latency lags Bandwidth", CACM, 2004
last RAM point based on DDR4-3400 RAM with 16-18-18-36 timings
later CPU points based on GHz + approx. pipeline depth of various AMD/Intel CPUs

# CPU/memory time per operation

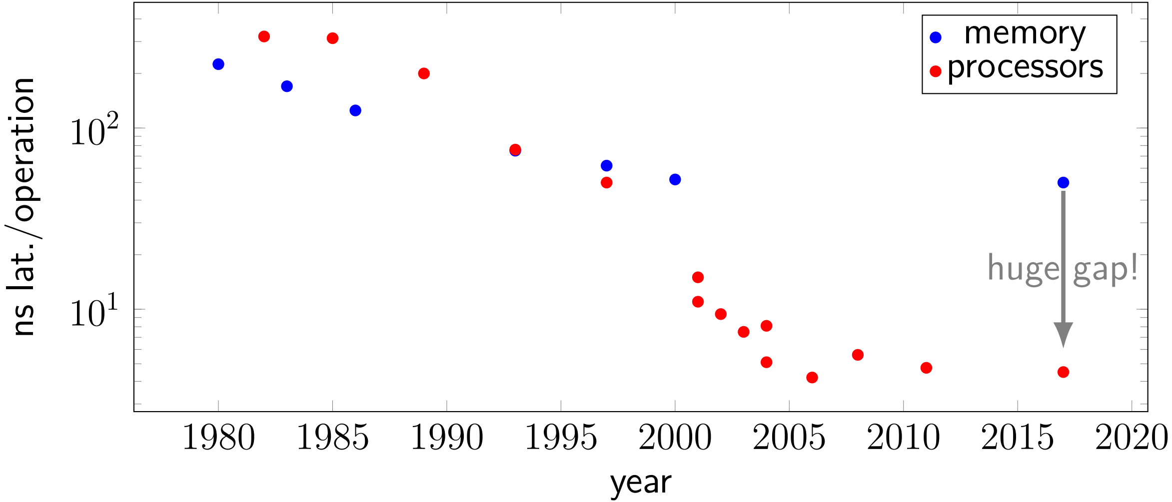
(everything approximate...)




data up to 2001, data via Patterson, "Latency lags Bandwidth", CACM, 2004
last RAM point based on DDR4-3400 RAM with 16-18-18-36 timings
later CPU points based on GHz + approx. pipeline depth of various AMD/Intel CPUs

# CPU/memory time per operation

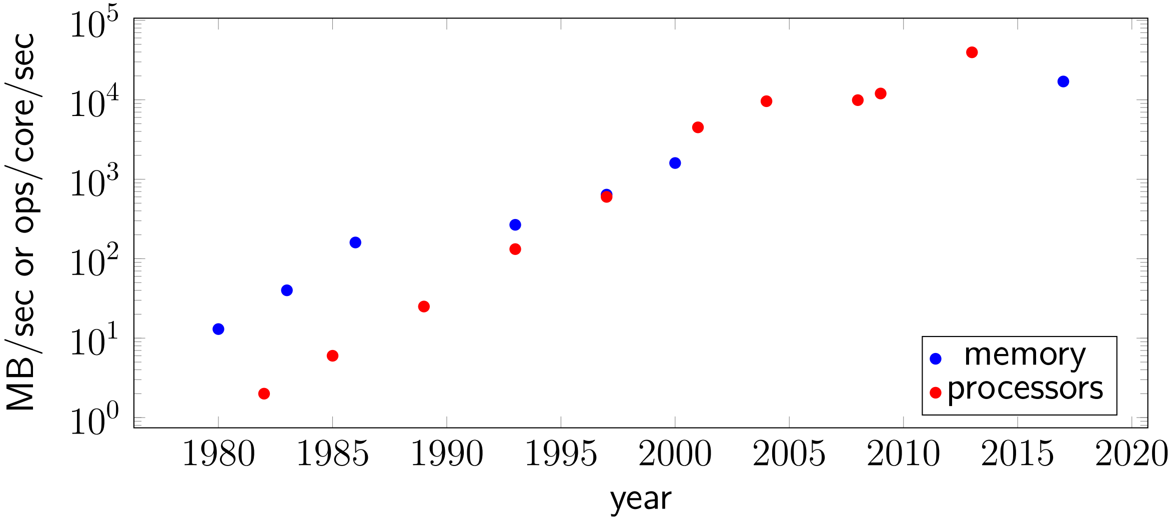
(everything approximate...)




data up to 2001, data via Patterson, "Latency lags Bandwidth", CACM, 2004
last RAM point based on DDR4-3400 RAM with 16-18-18-36 timings
later CPU points based on GHz + approx. pipeline depth of various AMD/Intel CPUs

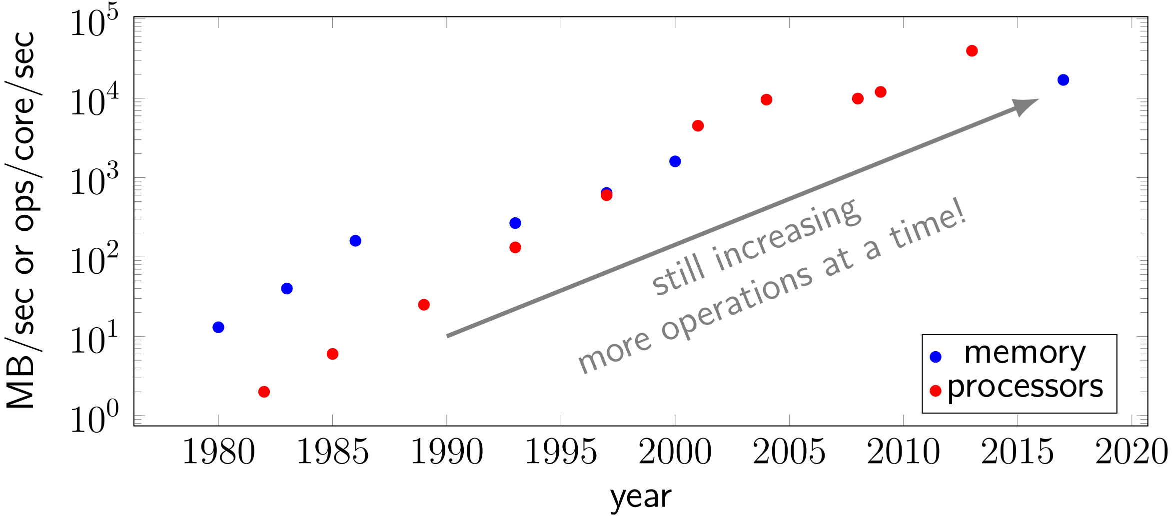
# CPU/memory processed per ns

(everything approximate...)

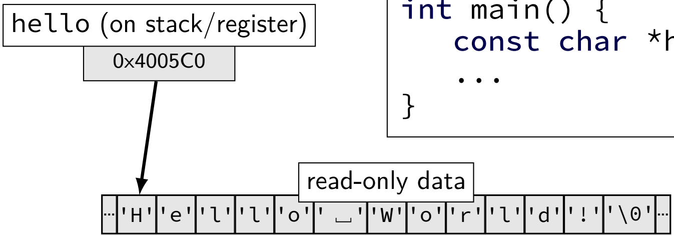


# CPU/memory processed per ns

(everything approximate...)



# strings in C




```c
int main() {
    const char *hello = "Hello␣World!";
    ...
}
```

# strings in C

```
int main() {
    const char *hello = "Hello␣World!";
    ...
}
```

hello (on stack/register)

0x4005C0

read-only data

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| … | 'H' | 'e' | 'l' | 'l' | 'o' | '␣' | 'W' | 'o' | 'r' | 'l' | 'd' | '!' | '\0' | … |

| | address(es) | value | |
|---|---|---|---|
| string (constant data) | 0x4005c0 | 0x48 | 'H' |
| | 0x4005c1 | 0x65 | 'e' |
| | 0x4005c2 | 0x6c | 'l' |
| | 0x4005c3 | 0x6c | 'l' |
| | 0x4005c4 | 0x6f | 'o' |
| | … | … | |
| | 0x4005ca | 0x21 | '!' |
| | 0x4005cb | 0x00 | '\0' |
| pointer (on stack) | … | … | |
| | 0x7fff3488-8f | 0x4005c0 | hello |

# C standard library functions

header file: `string.h`

`size_t strlen(const char* s)` — number of chars in s

`char *strcpy(char *s1, const char *s2)` — copy `s2` to `s1`, return `s1`

`char *strcat(char *s1, const char *s2)` — append `s2` to `s1`, return `s1`

# implementing strlen

```
size_t strlen(const char* s) {
    size_t i = 0;
    while (s[i] != '\0')
        i += 1;
    return i;
}
```

# a strcpy inquiry (1)

```
char *hello = "Hello!";
char *bye = "Bye!";
strcpy(bye, hello);
```

# a strcpy inquiry (1)

```
char *hello = "Hello!";
char *bye = "Bye!";
strcpy(bye, hello);
```

C result: Segmentation fault C++ result: compile error, `"Hello!"` is const

# a strcpy inquiry (2)

```
const char *hello = "Hello!";
char bye[5] = {'B', 'y', 'e', '!', '\0'}; // or "Bye!" (same e
strcpy(bye, hello);
```

# a strcpy inquiry (2)

```
const char *hello = "Hello!";
char bye[5] = {'B', 'y', 'e', '!', '\0'}; // or "Bye!" (same e
strcpy(bye, hello);
```

same as:

```
bye[0] = 'H'; bye[1] = 'e'; bye[2] = 'l'; bye[3] = 'l'; bye[4]
bye[5] = '!'; bye[6] = '\0';
```

goes out of bounds!

# a strcpy inquiry (3)

```
void foo() {
    const char *hello = "Hello!";
    char *dest = malloc(strlen(hello) + 1);
    strcpy(dest, hello);
    doSomethingWith(dest);
}
```

# a strcpy inquiry (3)

```
void foo() {
    const char *hello = "Hello!";
    char *dest = malloc(strlen(hello) + 1);
    strcpy(dest, hello);
    doSomethingWith(dest);
}
```

probably leaks memory

# strcat

```
const char *hello = "Hello,␣";
const char *world = "World!";
char *result = malloc(strlen(hello) + strlen(world) + 1);
strcpy(result, hello);
strcat(result, world);
```

# some code with memory leaks

```
// allocate a space in memory for result
char *result = malloc (sizeof (*result));
*result = '\0';
while (i < argc) {  // while there are still args
    char *s = malloc (sizeof (*s) *
            (strlen(result) + strlen(argv[i]) + 1));
    strcpy (s, result);
    strcat (s, argv[i]);
    result = s;
    i++;
}
printf ("Concatenation:␣%s\n", result);
```

# some code with memory leaks

```
// allocate a space in memory for result
char *result = malloc (sizeof (*result));
*result = '\0';
while (i < argc) {  // while there are still args
    char *s = malloc (sizeof (*s) *
            (strlen(result) + strlen(argv[i]) + 1));
    strcpy (s, result);
    strcat (s, argv[i]);
    free(result);
    result = s;
    i++;
}
printf ("Concatenation:␣%s\n", result);
```

exercise: why `result` and not `s`?

# some code with memory leaks

```
// allocate a space in memory for result
char *result = malloc (sizeof (*result));
*result = '\0';
while (i < argc) {  // while there are still args
    char *s = malloc (sizeof (*s) *
            (strlen(result) + strlen(argv[i]) + 1));
    strcpy (s, result);
    strcat (s, argv[i]);
    free(result);
    result = s;
    i++;
}
printf ("Concatenation:␣%s\n", result);
```

exercise: why `result` and not `s`?

# memory leak finding

idea: look at all pointers on stack, in global variables

and all pointers contained in objects those reference

and …

and compare to list of all allocated objects

done by tools like Valgrind Memcheck or AddressSanitizer

# recall: big-oh matters

not useful for fine-grained analysis

assumption: operations take the same amount of time

- caches? — not taken into account
- different versions of instructions
- …

# recursion to tail recursion

```
int factorial_recursive(int x) {
    if (x <= 1)
        return 1;
    else
        return x * factorial_recursive(x-1);
}
```

---

```
int factorial_tail_recursive(int x, int y = 1) {
    if (x <= 1)
        return y;
    else
        return factorial_tail_recursive(x-1, x*y);
}
```

# tail recursion: avoiding call

```
factorial_tail_recursive:
  cmp edi, 1
  jle .L4
.L2:
  imul esi, edi
  sub edi, 1
  jmp factorial_tail_recursive
  // same effect as:
  // call factorial_tail_recursive
  // ret
.L4:
  mov eax, esi
  ret
```

# tail recursion: avoiding call

```
factorial_tail_recursive:
  cmp edi, 1
  jle .L4
.L2:
  imul esi, edi
  sub edi, 1
  jmp factorial_tail_recursive
  // same effect as:
  // call factorial_tail_recursive
  // ret
.L4:
  mov eax, esi
  ret
```

# tail recursion

saves lots of stack space ($\Theta(x)$ space to $\Theta(1)$ space)

easier for compilers to do

"tail" requirement: must be last thing to do

…so it's okay to return directly to caller

# tail recursion: things on the stack

```
example_function:
    push rbx
    cmp rdi, 0
    je base_case
    ...
    ...
    pop rbx
    jmp example_function
base_case:
    pop rbx
    mov rax, ...
    ret
```

# tail recursion: things on the stack

```
example_function:
    push rbx
    cmp rdi, 0
    je base_case
    ...
    ...
    pop rbx
    jmp example_function
base_case:
    pop rbx
    mov rax, ...
    ret
```

# tail recursion to loop

```
int factorial_tail_recursive(int x, int y = 1) {
    if (x <= 1)
        return y;
    else
        return factorial_tail_recursive(x-1, x*y);
}
```

---

```
int factorial_loop(int x) {
    int y = 1;
    while (x > 1) {
        y *= x;
        x--;
    }
    return y;
}
```