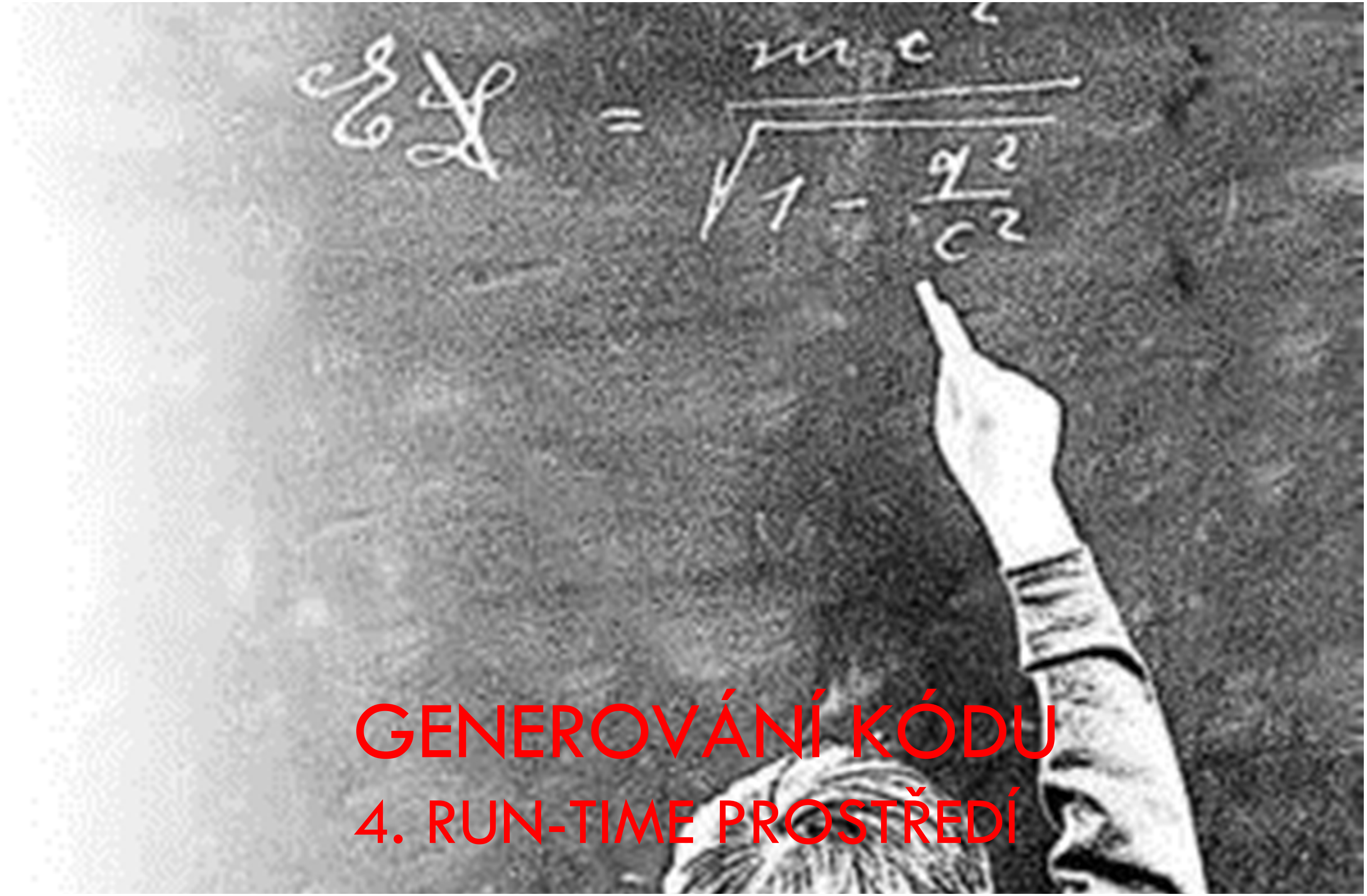


GENEROVÁNÍ KÓDU

4. RUN-TIME PROSTŘEDÍ

2011 Jan Janoušek

MI-GEN

INTRODUCTION

Compiler:

- implements the source language definitions,
- cooperates with the operating systém and other system sw,
- creates and manages a **run-time environment.**

Run-time environment deals with:

- the layout and allocation of storage locations of objects in the source program,
- linkages between procedures, passing parameters,
- …

When writing compiler, the code for the run-time environment must also be generated and the related issues must be considered.

NOTE. TYPICAL COMPUTER MEMORY HIERARCHY

Kind of memory	Typical access times	Typical sizes
Virtual memory (disks)	3 – 15 ms	> 16 GB
Physical memory	100 – 150 ns	156 MB – 16 GB
2nd level cache	40 – 60 ns	128KB – 4MB
1st level cache	5 – 10 ns	16 – 64 KB
Registers (processor)	1 ns	32 words

STORAGE ORGANIZATION

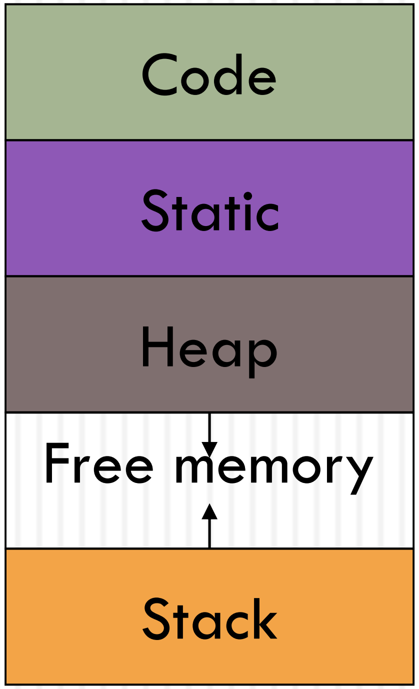
Usage of memory

- Typically, executing program runs in its own logical address space provided by the operating system. The operating system maps the logical addresses into physical addresses, which can be spread throughout memory.

- Source programming languages differ in using static and dynamic objects. Locations of static objects in memory can be known in the compile-time, whereas the locations of dynamic objects is known during run-time. There are languages:

  - with only static objects, e.g. Fortran 77
  - with both static and dynamic objects, e.g. C
  - with mostly dynamic objects, e.g. Lisp

Typical subdivision of run-time memory



Subdivision of run-time memory

Static part:

- Objects created when program is compiled, persists throughout run. Global constants, global and static variables, data generated by a compiler, …

- **Heap:**

  - Objects created/deleted in any order during run-time.
  - Heap contains dynamic variables and objects.
  - In some environments, heap is maintained by an automatic garbage collection.

Subdivision of run-time memory

- **Stack:**
  - Objects created/destroyed in last-in, first-out order.
  - Variables and objects local to a procedure (in so-called activation records). Stack supports call/return policy for procedures.

Allocation of static objects

Static objects

```
class Example {
    public static final int a = 3;
    public void hello() {
        System.out.println("Hello");
    }
}
```

- Static class variable
- Code for hello method
- String constant “Hello”
- Information about the Example class

Static objects

- Advantages:
  - Zero-cost memory management
  - Often faster access (address a constant)
  - No out-of-memory danger
- Disadvantages:
  - Size and number must be known beforehand

Allocation of space in the stack

Stack-Allocated Objects

- Natural for supporting recursion.
- Idea: some objects persist from when a procedure is called to when it returns.
- Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.
- Each invocation of a procedure gets its own frame (**activation record**) where it stores its own local variables and bookkeeping information.
- Calling a procedure is implemented by **calling sequence,** returning is implemented by **return sequence.**

What to save to activation record? example

(Real C)

```
int fib(int n) {
  if (n<2)
    return 1;
  else
    return
     fib(n-1) +
     fib(n-2);
}
```

(Assembly-like C)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n -1;
    tmp2 = fib(tmp1);
L2: tmp1 = n -2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 +    tmp3;
return tmp1;
}
```

What to save to activation record?

Programming languages differ:

- Languages without nested procedures, e.g. C
  - Simple: global variables are in the static storage. Local variables can be found in the local activation record on the top of the stack.

- Languages with nested procedures, e.g. Pascal, Lisp and functional languages in general (in Lisp a function can even create another function)
  - there must be some links among activation records.

A general activation record

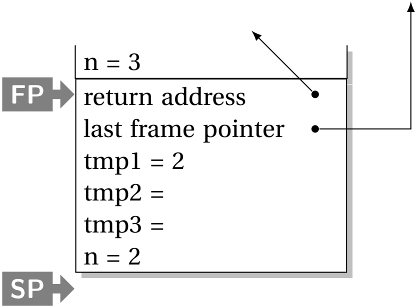
Actual parameters
Returned values
Control link (link to old activation record pointer)
Access link (link to higher nested procedure activation record)
Saved machine status (registers, etc…)
Local data (which are not placed in registers)
Temporaries (which are not placed in registers)

Example

Executing fib(3)


```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

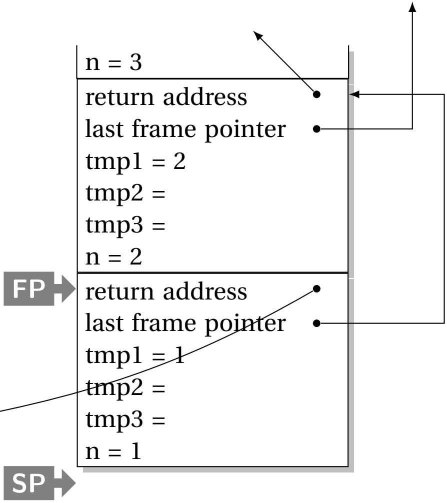
Executing fib(3)




```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

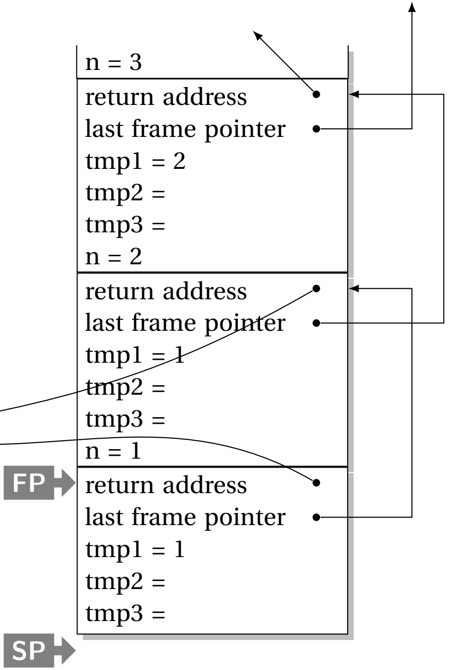
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



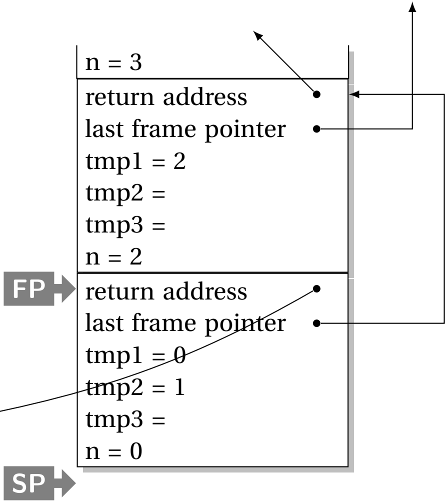
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



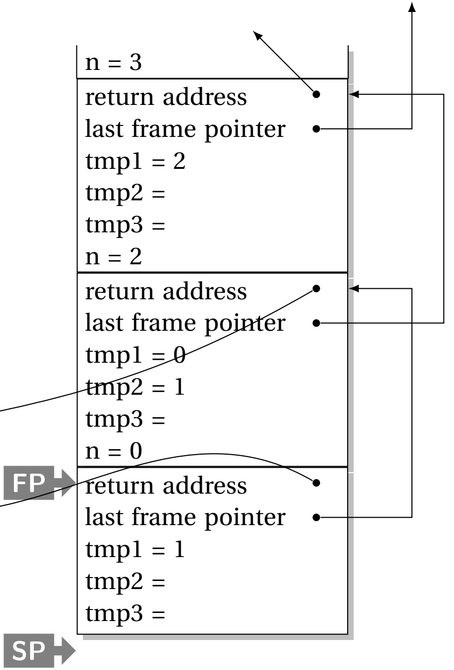
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



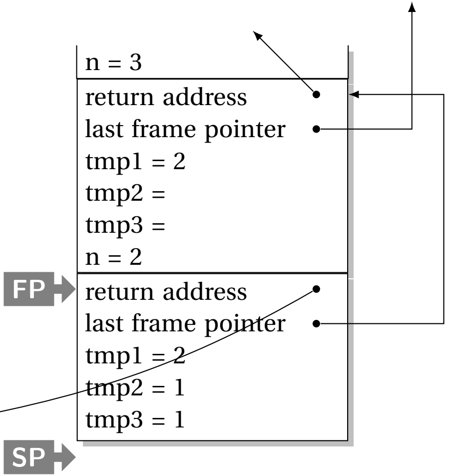
Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

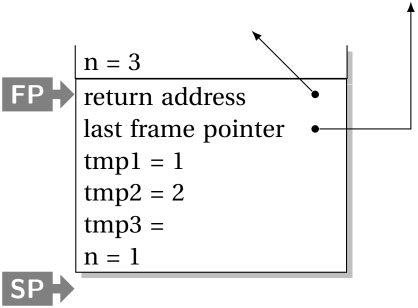


Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



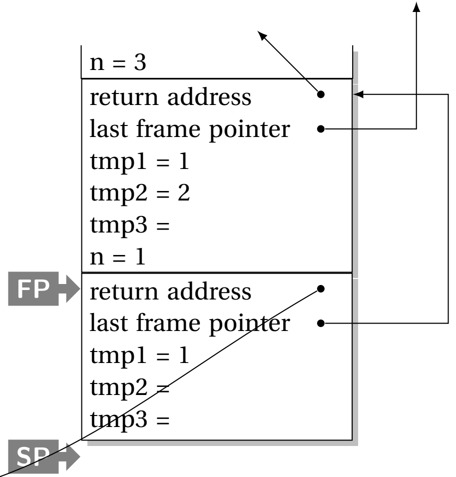
Executing fib(3)



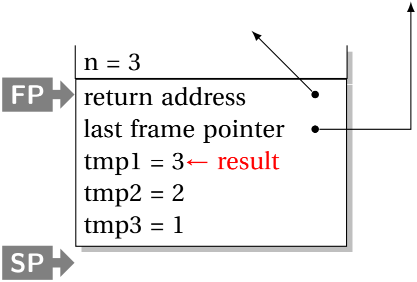

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

Executing fib(3)

```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```



Executing fib(3)




```
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

Passing parameters to procedures

- By value:
  - Actual value is computed and copied
  - Input parameter, behaves as local variable

- By reference:
  - Caller gives address in memory
  - Input/Output variables

- By name
  - Behaves as macro – the actual expression is inlined in the place of its use

Allocation of space in the heap

A *heap* is a region of memory where blocks can be allocated and deallocated in any order.

(These heaps are different than those in, e.g., heapsort)

Heap - example

Dynamic Storage Allocation in C

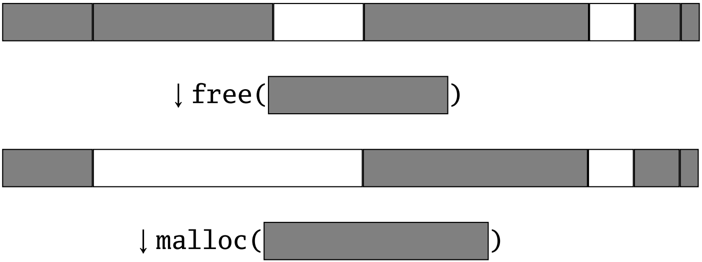
```c
struct point {
   int x, y;
};

int play_with_points(int n)
{
  int i;
  struct point *points;

  points = malloc(n * sizeof(struct point));

  for ( i = 0 ; i < n ; i++ ) {
    points[i].x = random();
    points[i].y = random();
  }

  /* do something with the array */

  free(points);
}
```

Dynamic Storage Allocation

Dynamic Storage Allocation

Dynamic Storage Allocation

Dynamic Storage Allocation



Dynamic Storage Allocation

Dynamic Storage Allocation

Rules:

- Each allocated block contiguous (no holes)
- Blocks stay fixed once allocated

`malloc()`

- Find an area large enough for requested block
- Mark memory as allocated

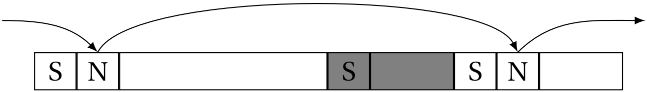
`free()`

- Mark the block as unallocated

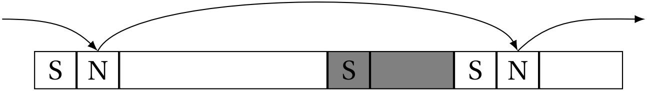
Simple Dynamic Storage Allocation

- Maintaining information about free memory
  - Simplest: Linked list
- The algorithm for locating a suitable block
  - Simplest: First-fit
- The algorithm for freeing an allocated block
  - Simplest: Coalesce adjacent free blocks

Simple Dynamic Storage Allocation

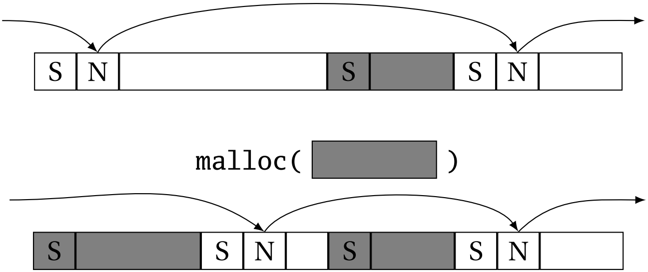


Simple Dynamic Storage Allocation

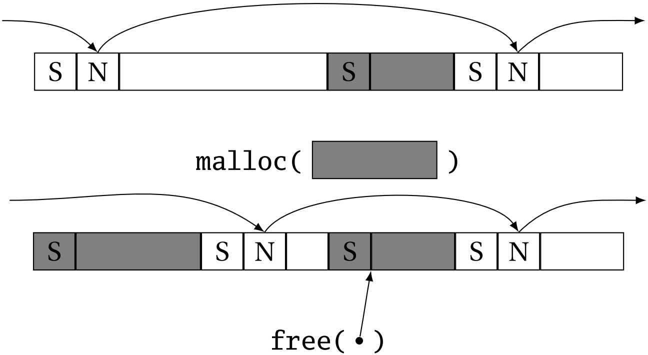


`malloc(`  `)`

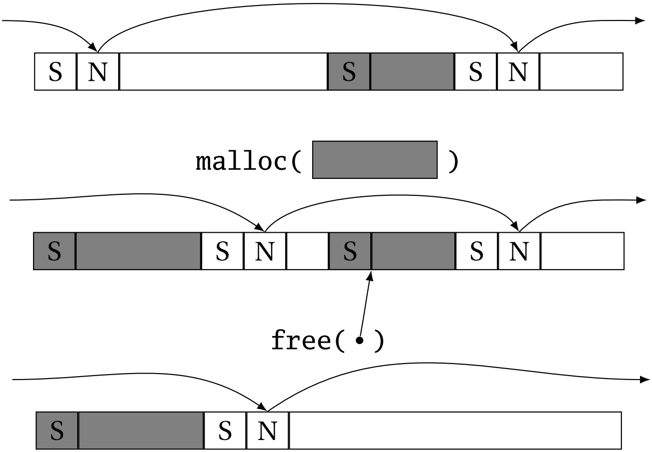
Simple Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Dynamic Storage Allocation

Many, many other approaches.

Other “fit” algorithms

Segregation of objects by size

More clever data structures

Heap Variants

Memory pools: Differently-managed heap areas

Stack-based pool: only free whole pool at once

- Nice for build-once data structures

Single-size-object pool:

- Fit, allocation, etc. much faster
- Good for object-oriented programs

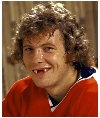
Fragmentation

`malloc(`) seven times give

`free()` four times gives

`malloc(`) ?

Need more memory; can't use fragmented memory.

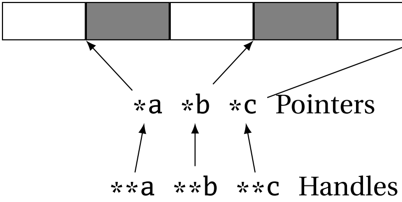


Hockey smile

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”

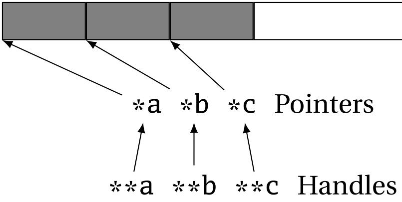


The original Macintosh did this to save memory.

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”



The original Macintosh did this to save memory.

Automatic garbage collection

- Remove the need for explicit deallocation.
- System periodically identifies reachable memory and frees unreachable memory.
- Reference counting approach.
- Many for example for curing fragmentation.