Recursion and Implementation of Functions

DEFINITION

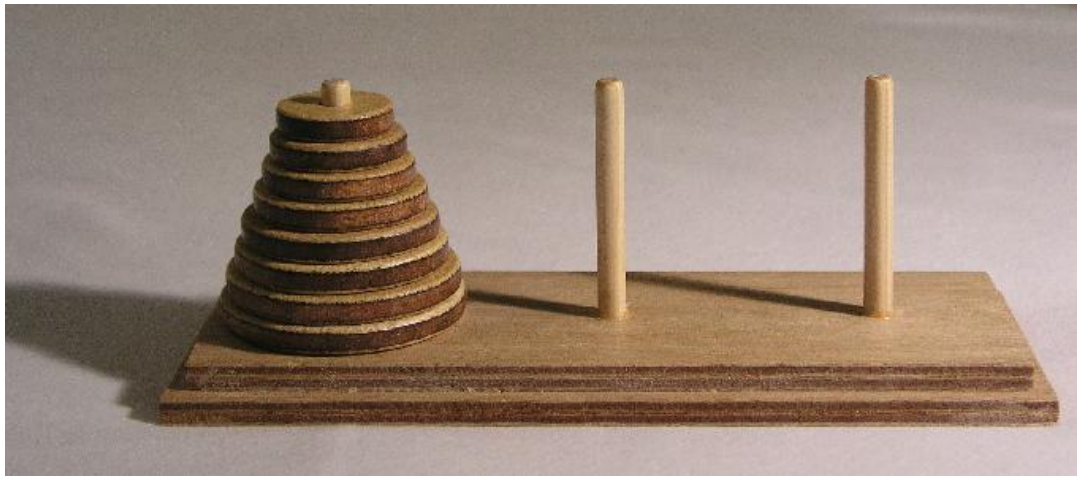
- *Recursive Function:–* a function that calls itself
  - Directly or indirectly
- Each recursive call is made with a new, independent set of arguments
  - Previous calls are suspended
- Allows very simple programs for very complex problems

Simplest Example

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

} // factorial
```

More Interesting Example
Towers of Hanoi



- Move stack of disks from one peg to another
- Move one disk at a time
- Larger disk may never be on top of smaller disk

Tower of Hanoi Program

```
#include <stdio.h>

void move (int n, int a, int c, int b);

int main() {
  int disks;
  printf ("How many disks?");
  scanf ("%d", &disks);

  move (disks, 1, 3, 2);

  return 0;
} // main
```

```
/* PRE:  n >= 0. Disks are arranged
  small to large on the pegs a, b, and
  c. At least n disks on peg a. No
  disk on b or c is smaller than the
  top n disks of a.
POST: The n disks have been moved from
  a to c. Small to large order is
  preserved. Other disks on a, b, c
  are undisturbed. */

void move (int n, int a, int c, int b)
  {
  if (n > 0)
    {
      move (n-1, a, b, c);
      printf ("Move one disk    from
  %d to %d\n", a, c);
      move (n-1, b, c, a);
```

- Is *pre-condition* satisfied before this call to **move**?

Tower of Hanoi Program

```
#include <stdio.h>

void move (int n, int a, int c, int b);

int main() {
  int disks;
  printf ("How many disks?");
  scanf ("%d", &disks);

  move (disks, 1, 3, 2);

  retu
} // ma
```

```
/* PRE:  n >= 0. Disks are arranged
  small to large on the pegs a, b,
  and c. At least n disks on peg a.
  No disk on b or c is smaller than
  the top n disks of a.
POST: The n disks have been moved
  from a to c. Small to large order
  is preserved. Other disks on a,
  b, c are undisturbed. */

void move (int n, int a, int c, int
  b) {
  if (n > 0)
    {
      move (n-1, a, b, c);
      printf ("Move one disk
from %d to %d\n", a, c);
      move (n-1, b, c, a);
    }    // if (n > 0)

  return;
} // move
```

- If *pre-condition* is satisfied here, is it still satisfied here?

And here?

Tower of Hanoi Program

```
#include <stdio.h>

void move (int n, int a, int c, int b);

int main() {
  int disks;
  printf ("How many disks?");
  scanf ("%d", &disks);

  move (disks, 1, 3, 2);

  return 0;
} // main
```

If *pre-condition* is true and if *n = 1*, does **move** satisfy the *post-condition?*

```
/* PRE:  n >= 0. Disks are arranged
  small to large on the pegs a, b,
  and c. At least n disks on peg a.
  No disk on b or c is smaller than
  the top n disks of a.
POST: The n disks have been moved
  from a to c. Small to large order
  is preserved. Other disks on a,
  b, c are undisturbed. */

void move (int n, int a, int c, int
  b) {
  if (n > 0)
    {
      move (n-1, a, b, c);
      printf ("Move one disk
  from %d to %d\n", a, c);
      move (n-1, b, c, a);
    }    // if

  return;
} // move
```

Can we reason that this program correctly plays *Tower of Hanoi*?

Why Recursion?

- Are articles of faith among CA students and faculty but …
- … a surprise to non-CA students.

- Some problems are *too hard* to solve without recursion
  - Most notably, the compiler!
  - Tower of Hanoi problem
  - Most problems involving linked lists and trees
    - (Later in the Data Structure course)

Recursion *vs.* Iteration

- Some simple recursive problems can be “unwound” into loops
  - But code becomes less compact, harder to follow!
- Hard problems cannot easily be expressed in non-recursive code
  - Tower of Hanoi
  - Robots or avatars that “learn”
  - Advanced games

Personal Observation

- From my own experience, programming languages, environments, and computer architectures that do not support recursion
- … are usually not rich enough to support a diverse portfolio of applications
  - I.e., a wide variety of problems in many different disciplines

QUESTIONS?

Implementing Recursion — *The Stack*

- Definition – *The Stack*
  - A *last-in, first-out* data structure provided by the operating system for *each* running program
  - For *temporary* storage of automatic variables, arguments, function results, and other stuff

- I.e., the working storage for *each, separate function call*.

The Stack (continued)

- *Every single time* a function is called, an area of the stack is reserved *for that particular call.*
- Known as its *activation record* in compiler circles.

Recursion is *so important* …

- … that all modern computer architectures specifically support it
  - Stack register
  - Instructions for manipulating *The Stack*
- … most modern programming languages allow it
  - But not Fortran and not Cobol

Recursion in *C*

- Parameters, results, and automatic variables allocated *on the stack*.
- Allocated when function or compound statement is entered
- Released when function or compound statement is exited
- Values *are not retained* from one call to next (or among recursions)

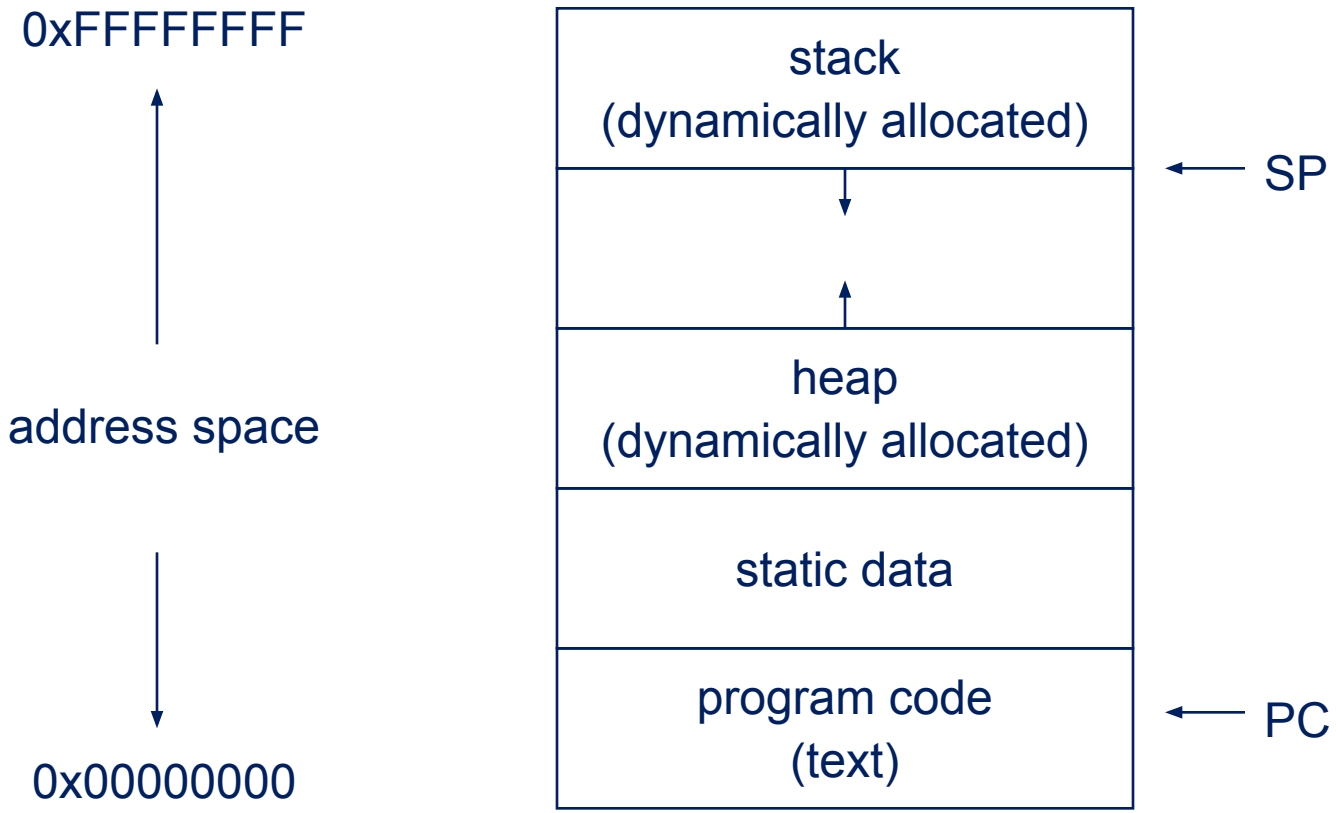
Arguments and Results

1. Space for *storing result* is allocated by caller
   - On *The Stack*
   - Assigned by **`return`** statement of function
   - For use by caller
2. *Arguments* are *values* calculated by caller of function
   - Placed on *The Stack* by caller in locations set aside for the corresponding parameters
   - Function may assign new value to parameter, but …
   - …caller *never* looks at parameter/argument values again!
3. Arguments are removed when callee returns
   - Leaving only the *result* value for the caller

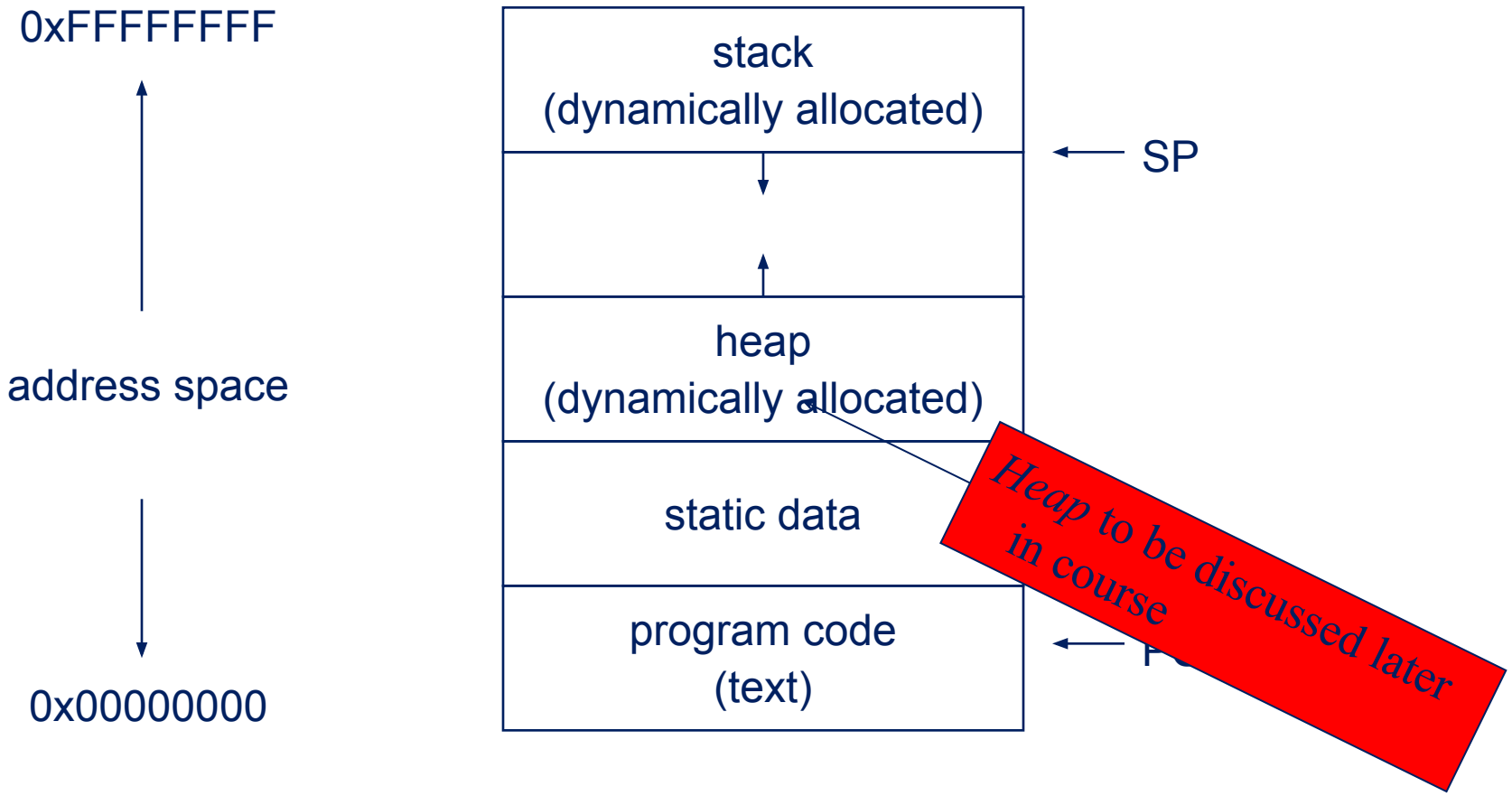
Typical Implementation of *The Stack*

- Linear region of memory
- *Stack Pointer* "growing" downward
- Each time some information is *pushed* onto The Stack, pointer moves downward
- Each time info is *popped* off of The Stack, pointer moves back upward

Typical Memory for Running Program (Windows & Linux)



Typical Memory for Running Program (Windows & Linux)



How it works

- Imagine the following program:–

```
int factorial(int n){
  …
  /* body of factorial function */
  …
} // factorial
```

- Imagine also the caller:–

```
int x = factorial(100);
```

- What does compiled code look like?

COMPILED CODE: THE *CALLER*

```
int x = factorial(100);
```

- Provide integer-sized space on stack for result, so that *calling function* can find it
- Evaluate the expression "100" and leave it on the stack (after result)
- Put the current program counter somewhere so that *factorial* function can return to the right place in *calling* function
- Transfer control to the called function

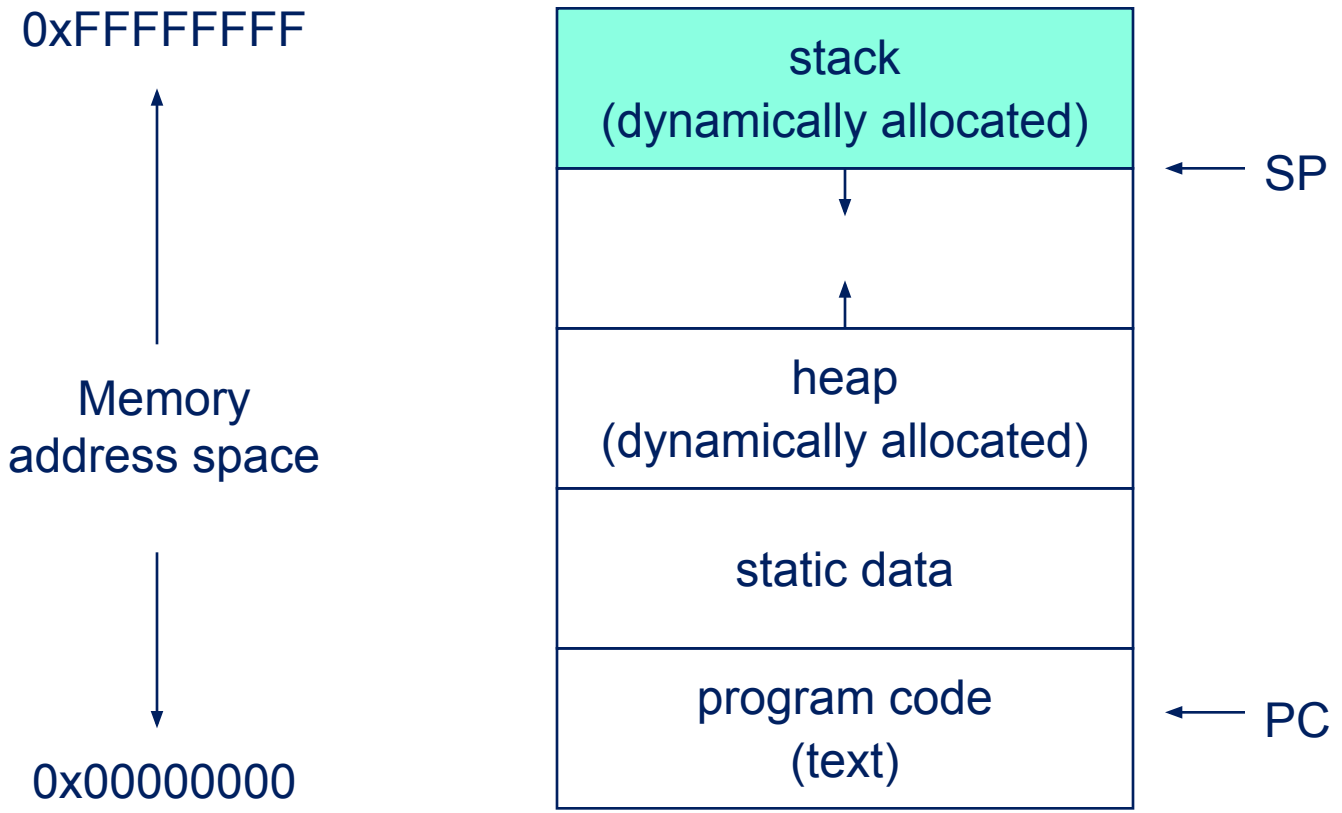
COMPILED CODE: *FACTORIAL* FUNCTION

- Save the *caller*'s registers in a dedicated space in the activation record
- Get the parameter *n* from the stack
- Set aside some memory for local variables and intermediate results on the stack

- Do whatever *factorial* was programmed to do

- Put the result in the space allocated by the *caller*
- Restore the *caller*'s registers
- Transfer back to the program counter saved by the *caller*

Typical Address Space (Windows & Linux)



NOTE

- Through the magic of operating systems, each running program has its *own* memory
  - Complete with stack & everything else

- Called a *process*

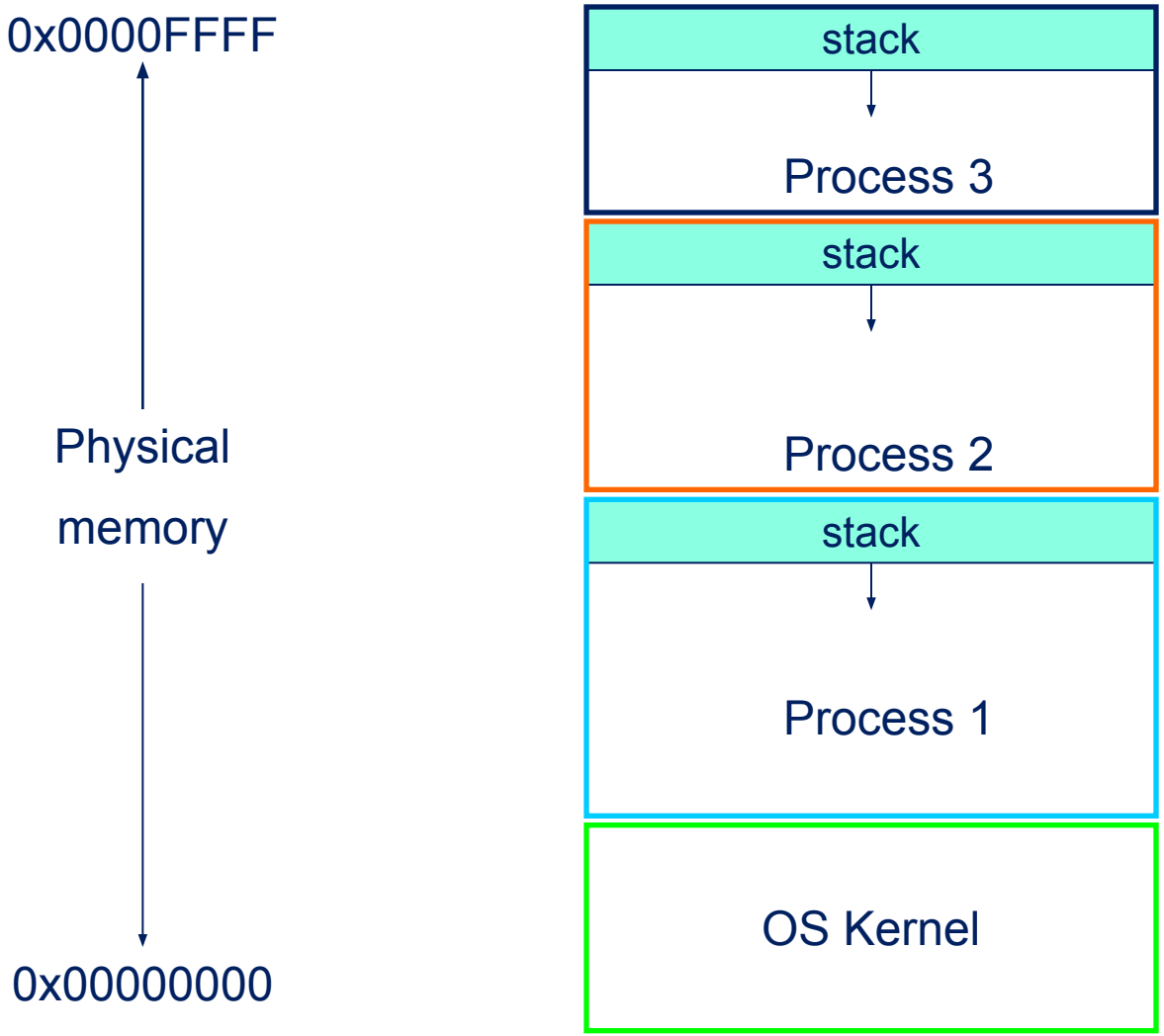
Windows, Linux, Unix, etc.

NOTE (CONTINUED)

- Not necessarily true in small, embedded systems
  - Real-time & control systems
  - Mobile phone & PDA
  - Remote sensors, instrumentation, etc.

- Multiple running programs *share* a memory
  - Each in own partition with own stack
  - Barriers to prevent each from corrupting others

Shared Physical Memory



QUESTIONS?

The Stack (summary)

- The stack gives each function *call* its own, private place to work
  - Separate from all other calls to same function
  - Separate from all calls to other functions
  - Place for automatic variables, parameters, results

The Stack (continued)

- Originally intended to support recursion
  - Mostly for computer scientists
  - Compiler writers, etc.
- Powerful enabling concept
  - Allows function to be shared among multiple running programs
  - Shared libraries
  - Concurrent activities within a process

QUESTIONS?