C Stream Processing

CSE 333 Winter 2021

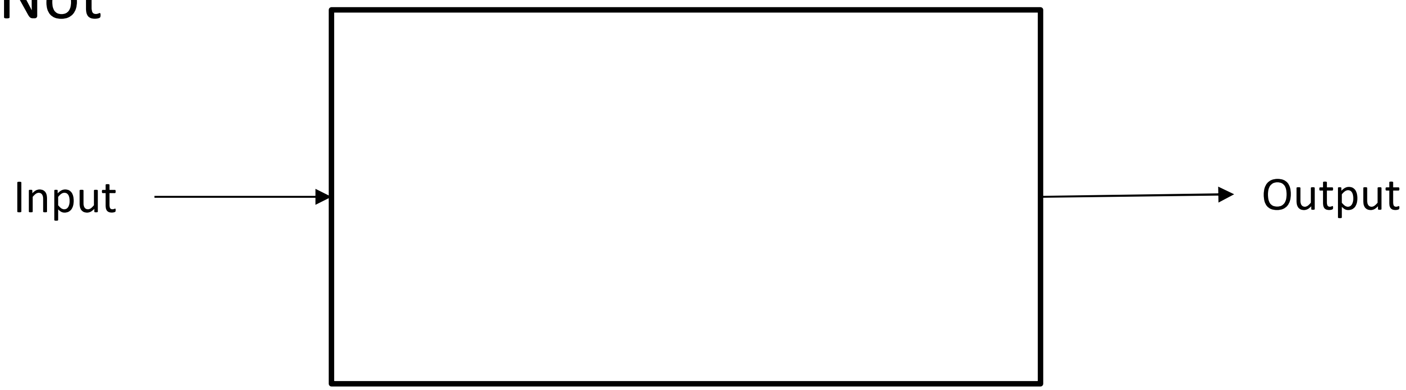
Instructor: John Zahorjan

Teaching Assistants:

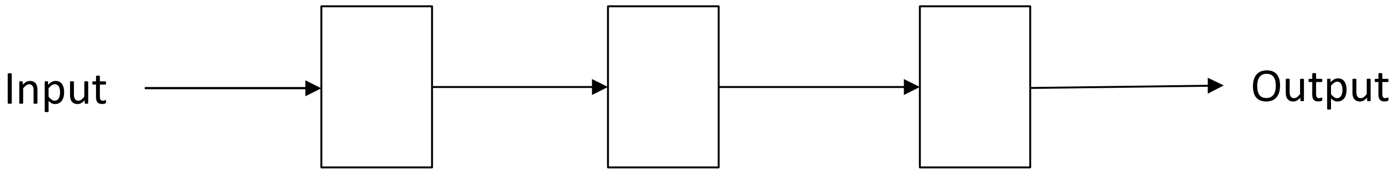
Matthew Arnold
Elizabeth Haker
Leo Liao
Guramrit Singh
Nonthakit Chaiwong
Henry Hung
Tim Mandzyuk
Jacob Cohen
Chase Lee
Benjamin Shmidt

Stream Processing Design

- Not



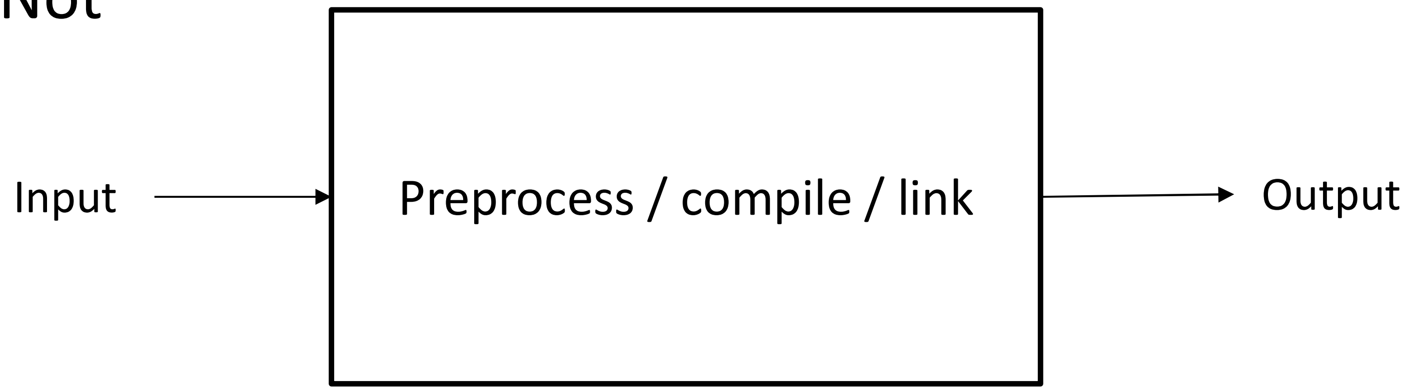
- Instead, many smaller pieces



"filters"

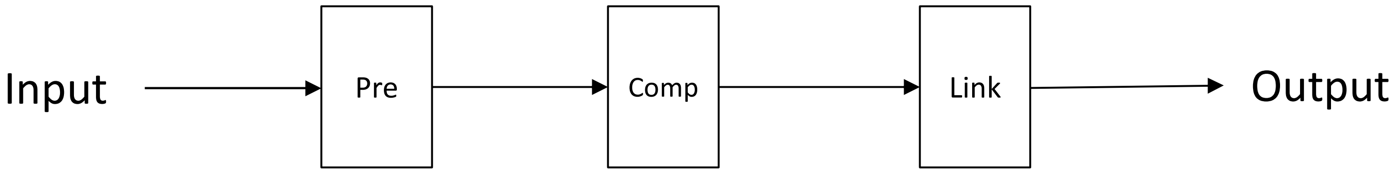
Example: gcc

- Not



Input → Preprocess / compile / link → Output

- Instead, many smaller pieces



Input → Pre → Comp → Link → Output

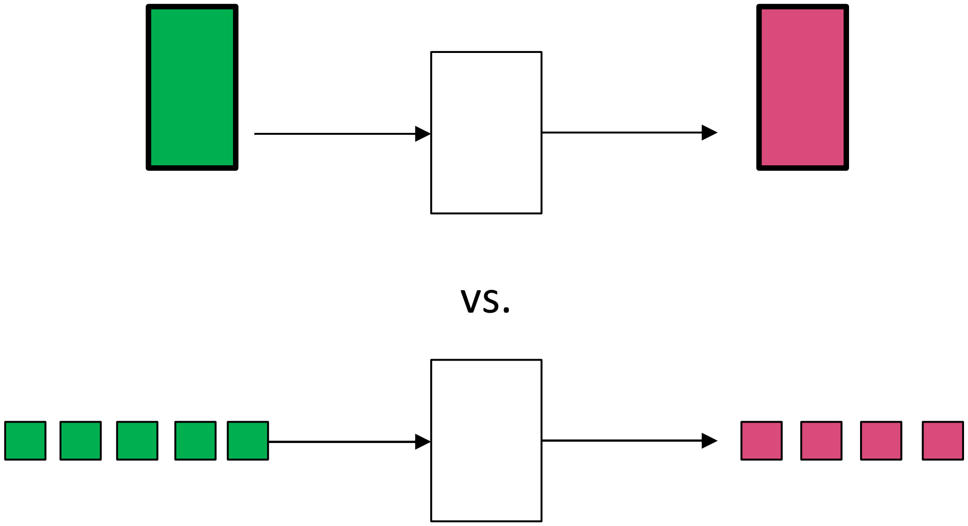
Components / Apps

- We try to build components that are re-usable
- We build apps by composing components
- The operating system and shell provide mechanisms that help with this
- **Output redirection**
  comp_1 >some_file

  Input Redirection
  comp_2 <some_file
- **Pipes**
  comp_1 | comp_2 | comp_3
- This view/style admittedly is most obviously compelling when data can be represented using text

Processing a Stream

- A stream is a linear flow of data
  - Process the data as it arrives, rather than reading all data before processing



  - The C compiler was originally, at least, able to stream process
    - Declare before use…

Data sources and streams

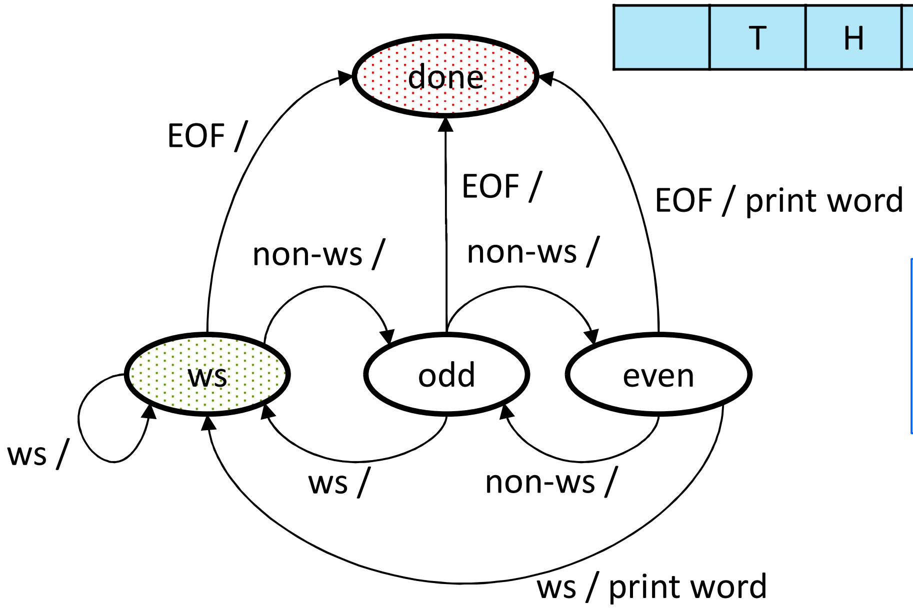
- Keyboard
  - A sequence of keystrokes
  - Usually no data available to your program until user hits enter
    - There’s a way for you to ask OS to pass every key press on to you
- Files
  - Arrays of bytes
  - Overwhelming default is to read bytes in order
    - Can jump around in bytes if you must
- Network
  - It’s a wire, or it’s a radio way
  - Data arrives a bit at a time, in order
  - “Conversations” are sequences of messages
- Graphical interfaces
  - A sequence of clicks

State Machines and Streams

- “State machines” are often useful abstractions for stream processing
  - Application is in some state
  - Each input token causes a state transition
  - Associated with each transition is some action
  - (CSE 311)

- The state machine’s input is a sequence of symbols
  - … a stream

Graphical Representation of State Machine

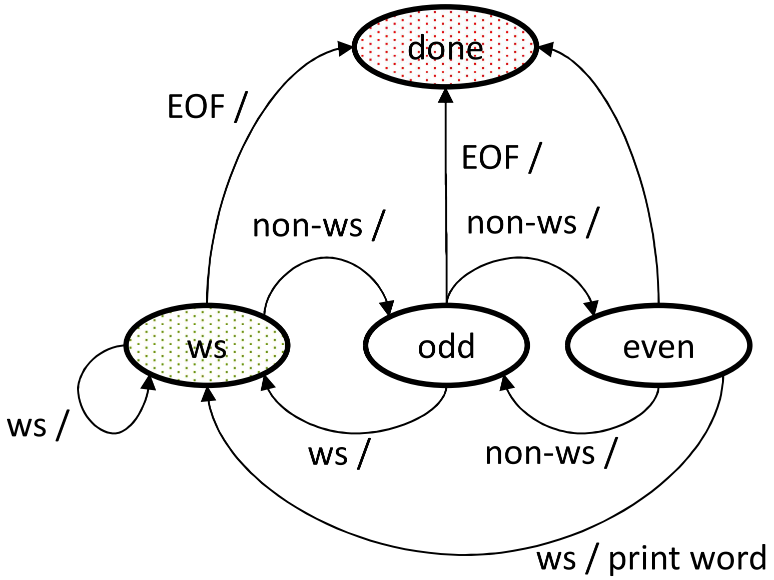


	T	H	R	E	E		F	O	U	R

Goal: print all "words" in input stream that have an odd number of characters

- Read input one character at a time
- Classify each input character
  - whitespace (ws) or non-whitespace (non-ws) or EOF
- In each state there is a transition for each input token type (symbol)
- Each transition identifies the next state and an optional action

Matrix Representation of State Machine



Input Symbol

Current State

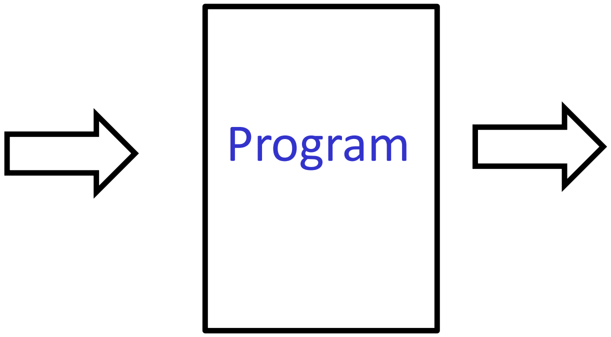
	ws	non-ws	EOF
ws	ws	odd	done
odd	ws	even	done
even	ws	odd	done

Transition Matrix

Larger Example: Words with all vowels in order

/usr/share/dict/words

```
...
abiogenous
abiogeny
abiological
abiologically
abiology
abioses
abiosis
abiotic
abiotical
abiotically
abiotrophic
abiotrophy
Abipon
...
```



Program Output

```
abietineous
abstemious
abstemiously
abstemiousness
abstentious
...
```

"Undisciplined Implementation"

- 1st problem – how to accumulate characters in a word?

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

#define MAX_WORD_SIZE 100
char word[MAX_WORD_SIZE];
int word_index = 0;

void addChar(char c)
{
  word[word_index] = c;
  word_index = (word_index+1) % MAX_WORD_SIZE;
}
```

How good an idea is this?

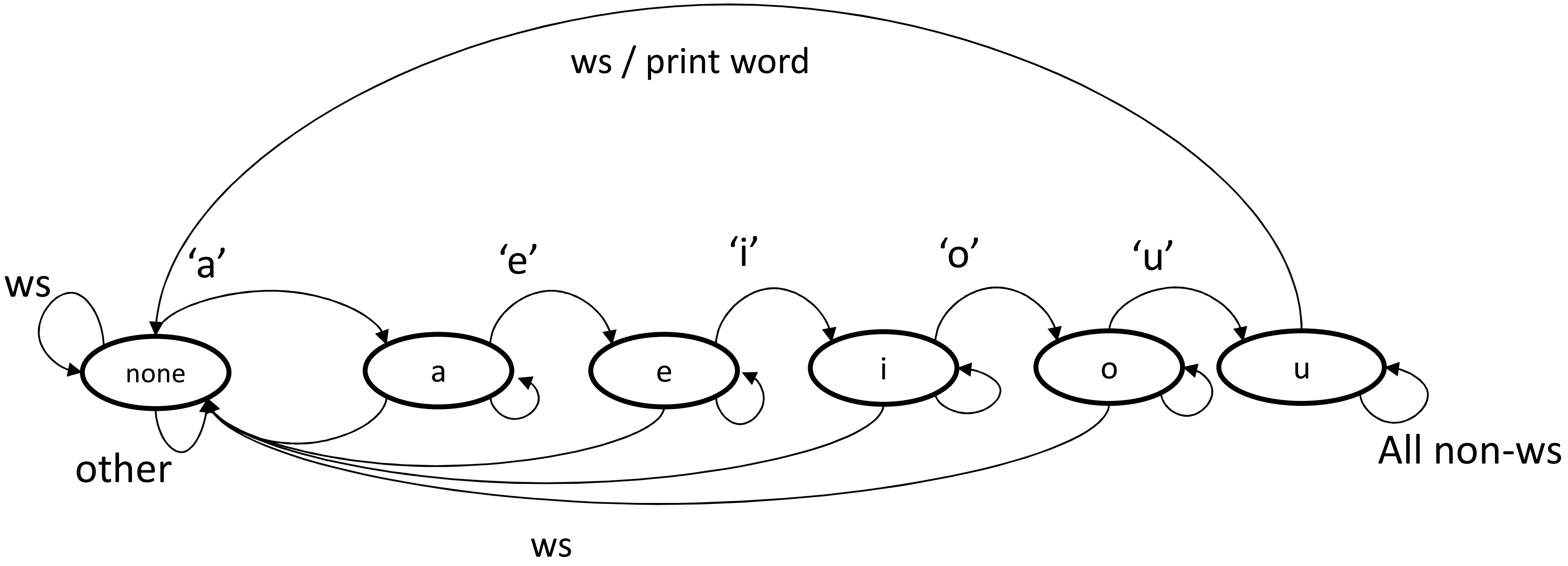
Undisciplined (cont.)

```
int main(int argc, char *argv[])
{
 char next_char = getchar();
 while ( next_char != EOF )
  {
   word_index = 0;
   while ( isspace(next_char) )
    next_char = getchar();
   while ( !isspace(next_char) && tolower(next_char) != 'a' && next_char != EOF )
    {
     addChar(next_char);
     next_char = getchar();
    }
   while ( !isspace(next_char) && tolower(next_char) != 'e' && next_char != EOF )
     {
      addChar(next_char);
      next_char = getchar();
     }
  ...
```

Undisciplined (last)

```
 ...
while ( !isspace(next_char) && tolower(next_char) != 'u' && next_char != EOF )
          {
            addChar(next_char);
            next_char = getchar();
          }
if ( tolower(next_char) == 'u'
        {
          while ( !isspace(next_char) && next_char != EOF )
            {
              addChar(next_char);
              next_char = getchar();
            }
          // print_word
          addChar('\0');
          printf("%s\n", word);
        }
 }  // end of while (next_char != EOF) loop
```

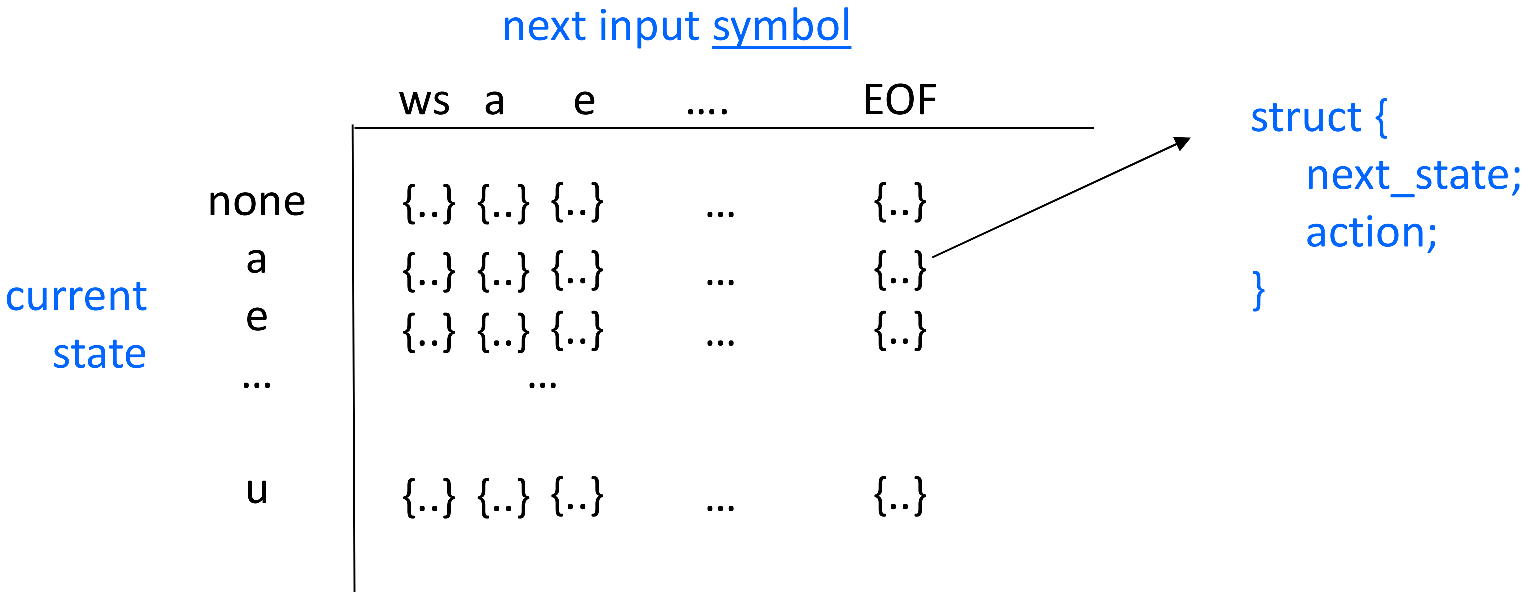
State Machine Version



I haven't shown all actions.
Additionally, EOF in any state signals end of execution.

State Machine Implementation

- Step 1
  - Represent state machine as a matrix



next input symbol

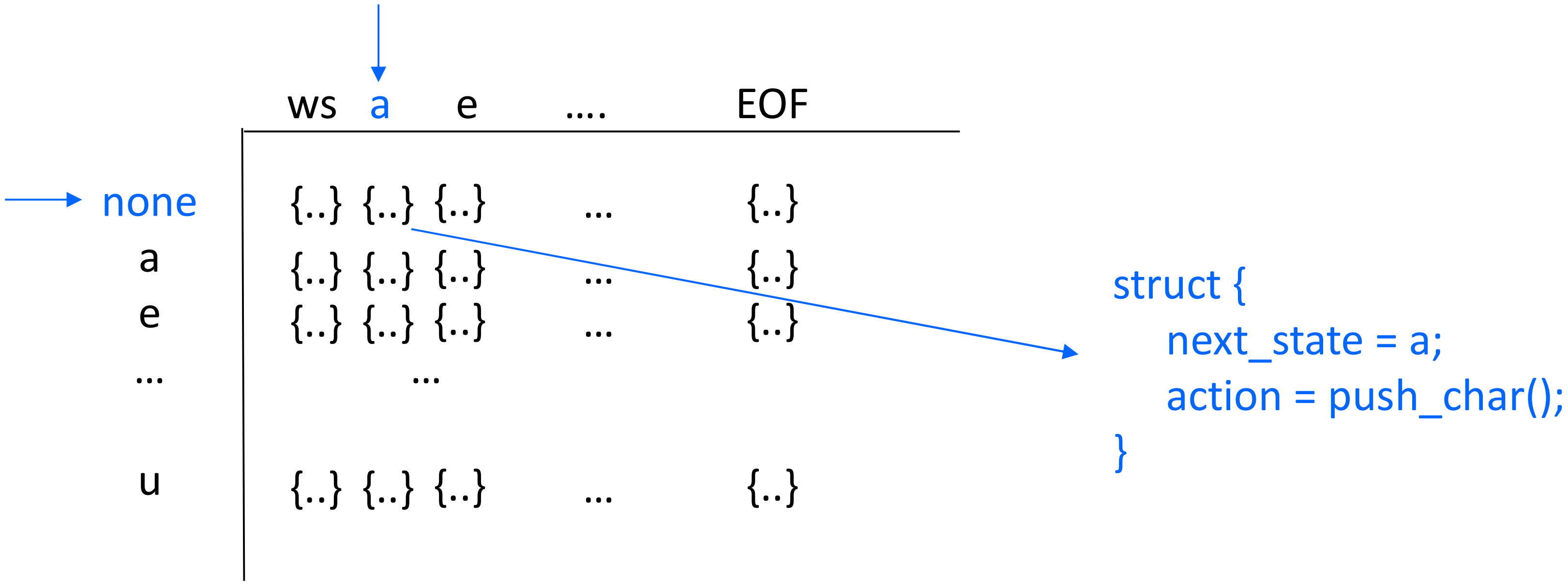
current state

	ws	a	e	….	EOF
none	{..}	{..}	{..}	…	{..}
a	{..}	{..}	{..}	…	{..}
e	{..}	{..}	{..}	…	{..}
…		…			
u	{..}	{..}	{..}	…	{..}

```
struct {
    next_state;
    action;
}
```

Example input: “anew ”

- current_state = none
  input token = a

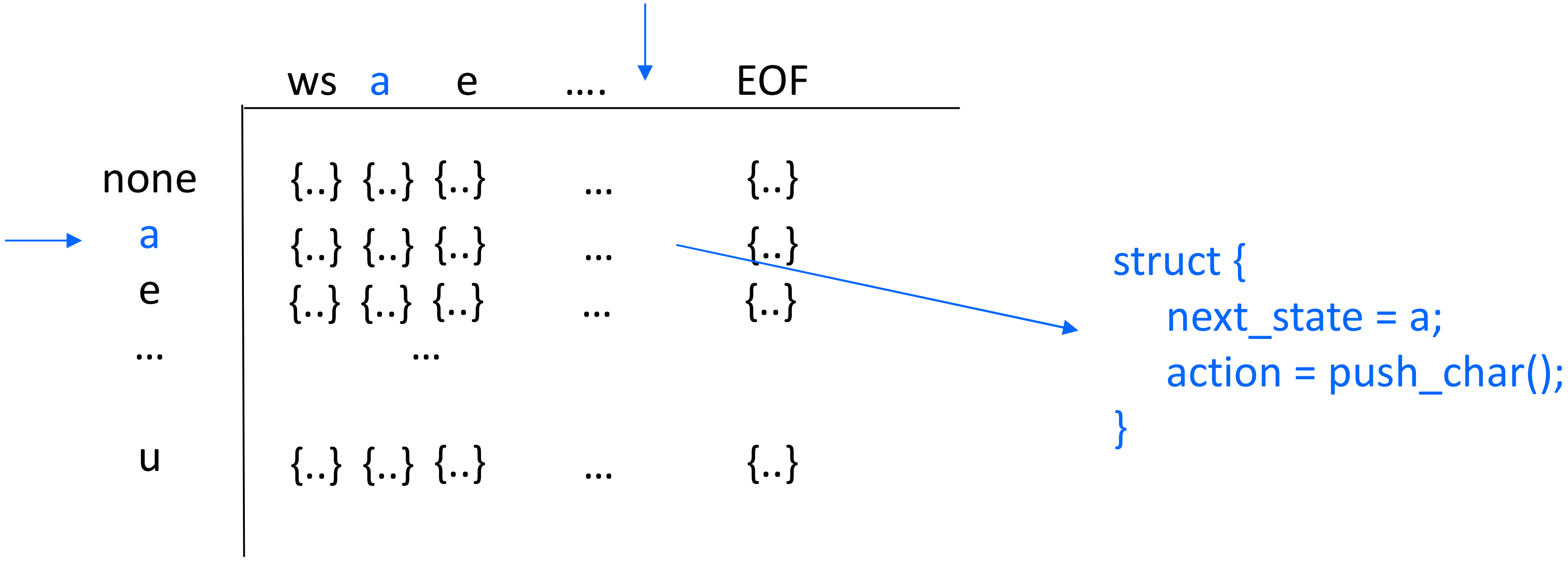


	ws	a	e	….	EOF
none	{..}	{..}	{..}	…	{..}
a	{..}	{..}	{..}	…	{..}
e	{..}	{..}	{..}	…	{..}
…		…			
u	{..}	{..}	{..}	…	{..}

```
struct {
    next_state = a;
    action = push_char();
}
```

Example input: “anew ”

- current_state = a
  input token = n

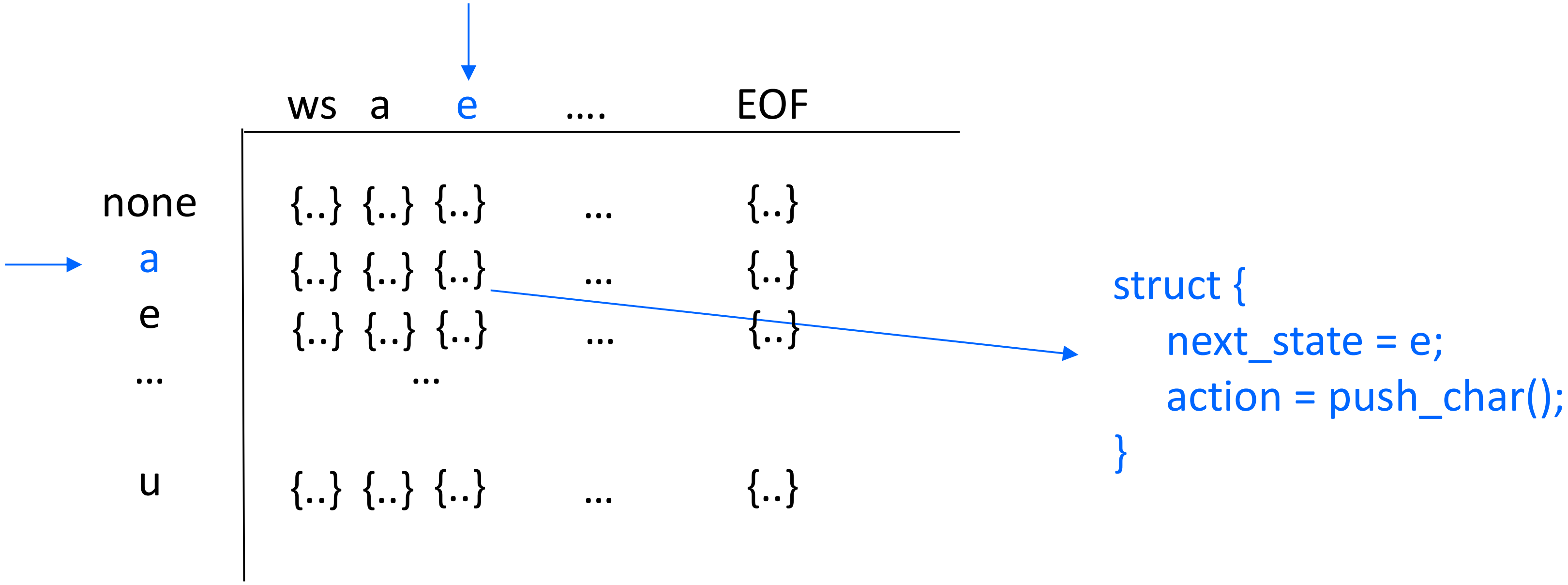


	ws	a	e	....	EOF
none	{..}	{..}	{..}	…	{..}
a	{..}	{..}	{..}	…	{..}
e	{..}	{..}	{..}	…	{..}
…		…			
u	{..}	{..}	{..}	…	{..}

```
struct {
    next_state = a;
    action = push_char();
}
```

Example input: “anew ”

- current_state = a
  input token = e



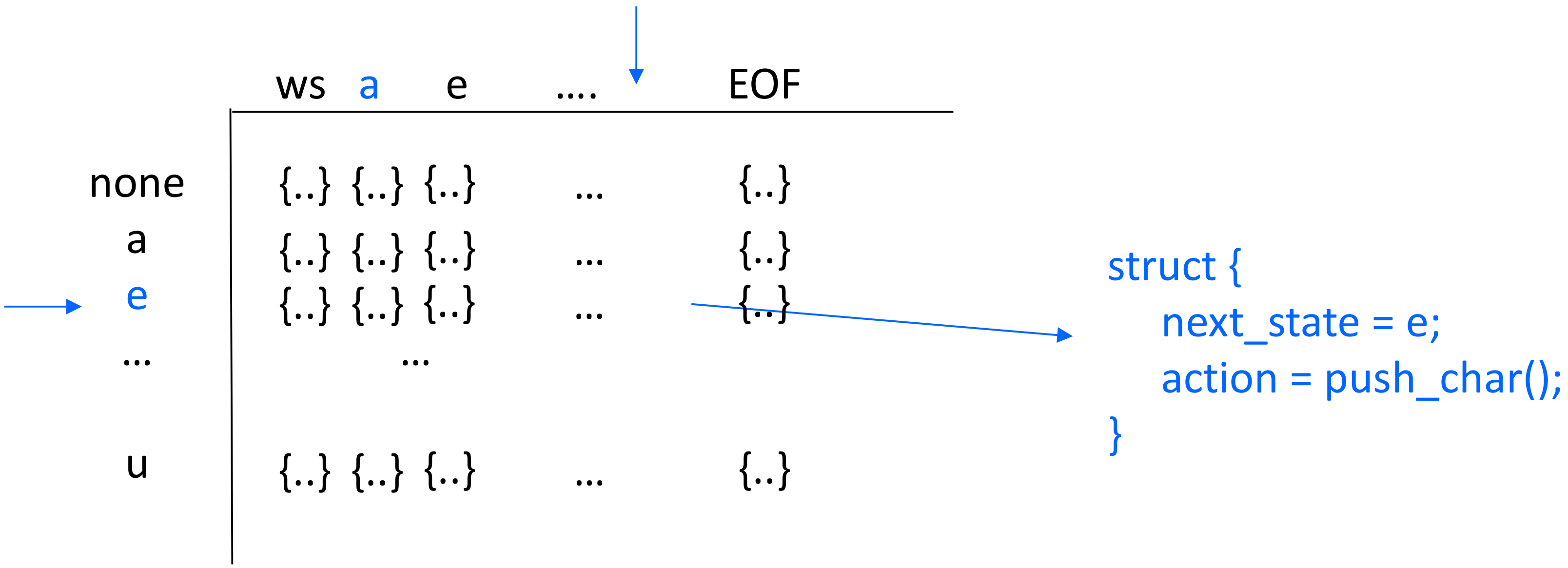
	ws	a	e	….	EOF
none	{..}	{..}	{..}	…	{..}
a	{..}	{..}	{..}	…	{..}
e	{..}	{..}	{..}	…	{..}
…		…			
u	{..}	{..}	{..}	…	{..}

```
struct {
    next_state = e;
    action = push_char();
}
```

Example input: “anew ”

- current_state = e

  input token = w

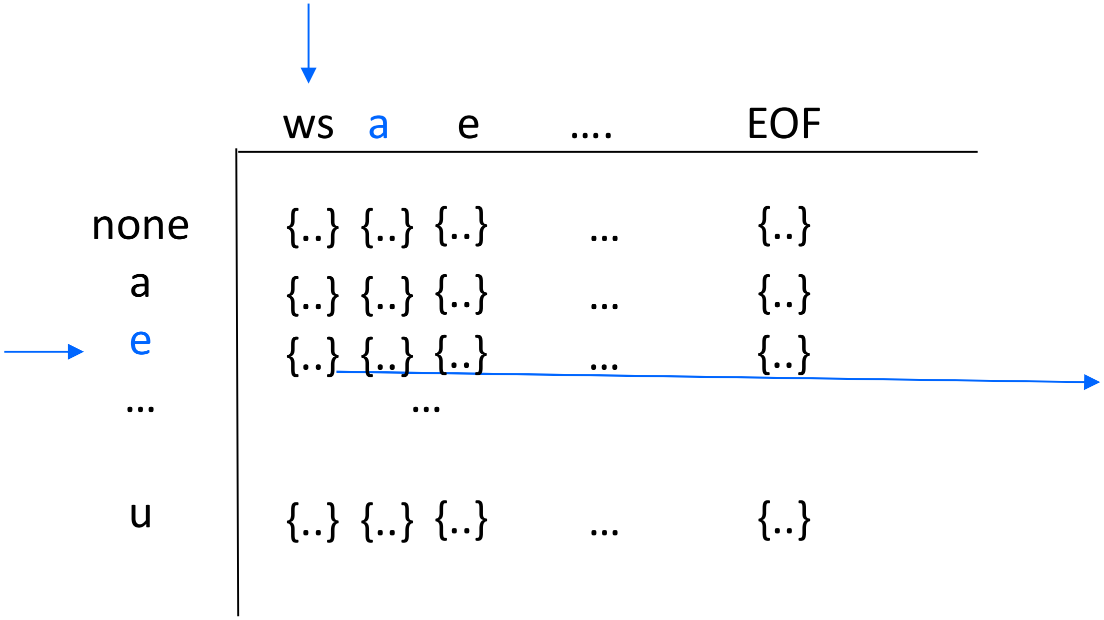


	ws	a	e	….	EOF
none	{..}	{..}	{..}	…	{..}
a	{..}	{..}	{..}	…	{..}
e	{..}	{..}	{..}	…	{..}
…		…			
u	{..}	{..}	{..}	…	{..}

```
struct {
    next_state = e;
    action = push_char();
}
```

Example input: “anew ”

- current_state = e
  input token = ‘ ‘



	ws	a	e	….	EOF
none	{..}	{..}	{..}	…	{..}
a	{..}	{..}	{..}	…	{..}
e	{..}	{..}	{..}	…	{..}
…		…			
u	{..}	{..}	{..}	…	{..}

```
struct {
    next_state = none;
    action = reset_word();
}
```

Implementation in C

- Example code will be pushed to your repository

```
typedef enum state_enum
 {
  state_none,
  state_a,
  state_e,
  state_i,
  state_o,
  state_u,
  NUM_STATES,
  state_done
 } State;
```

```
typedef enum char_class_enum
 {
  char_ws,
  char_a,
  char_e,
  char_i,
  char_o,
  char_u,
  char_non_ws,
  char_EOF,
  NUM_CHAR_CLASS
 } CharClass;
```

State Machine is Data

```
typedef void (*ActionFunction)(System*);
```

```
typedef struct transition_t {
  State next_state;
  ActionFunction action;
} Transition;
```

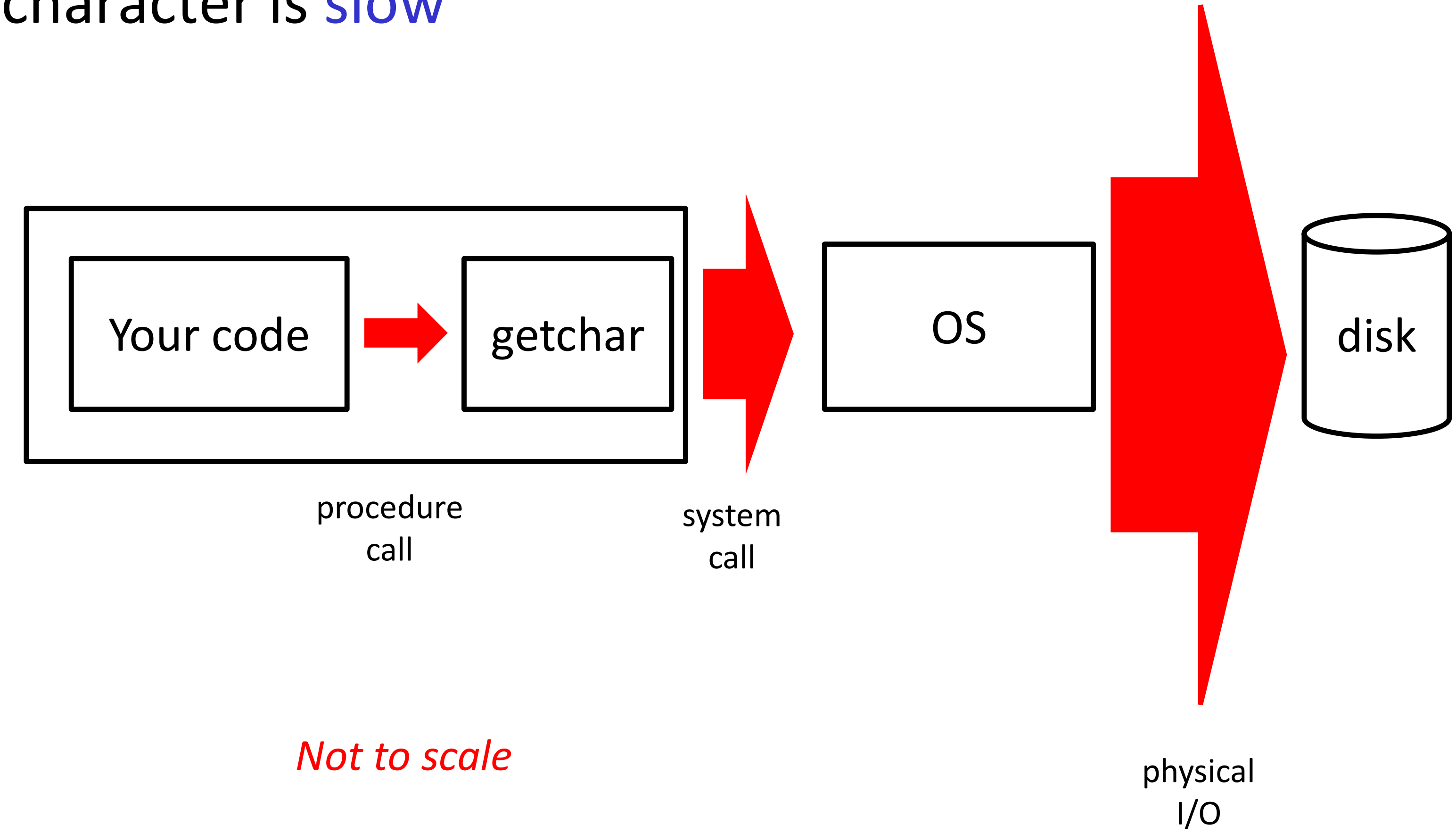
```
Transition transition_matrix[NUM_STATES][NUM_CHAR_CLASS] =
 {
  { // state_none
   {state_none, NULL},                  // char_ws
   {state_a,    System_addchar},        // char_a
   {state_none, System_addchar},  // char_e
   {state_none, System_addchar},  // char_i
   {state_none, System_addchar},  // char_o
   {state_none, System_addchar},  // char_u
   {state_none, System_addchar},  // char_non_ws
   {state_done, NULL },                 // char_eof
  },
  { // state_a
   {state_none, System_resetWord},
   {state_a,    System_addchar},
   {state_e,    System_addchar},
   {state_a,    System_addchar},
   {state_a,    System_addchar},
   {state_a,    System_addchar},
   {state_a,    System_addchar},
   {state_done, NULL}
  },
…
```

Complexity is Encoded in Data (Matrix)

```
int main(int argc, char *argv[]) {
  System system;
  ActionFunction action;
  System_initialize(&system);
 while (system.current_state != state_done ) {
     system.next_char = getchar();
     system.next_char_class = classifyChar(system.next_char);
     action = transition_matrix[system.current_state][system.next_char_class].action;
     if ( action != NULL )   action(&system);
     system.current_state =
                          transition_matrix[system.current_state][system.next_char_class].next_state;
  }
  return EXIT_SUCCESS;
}
```

Support for Stream Processing

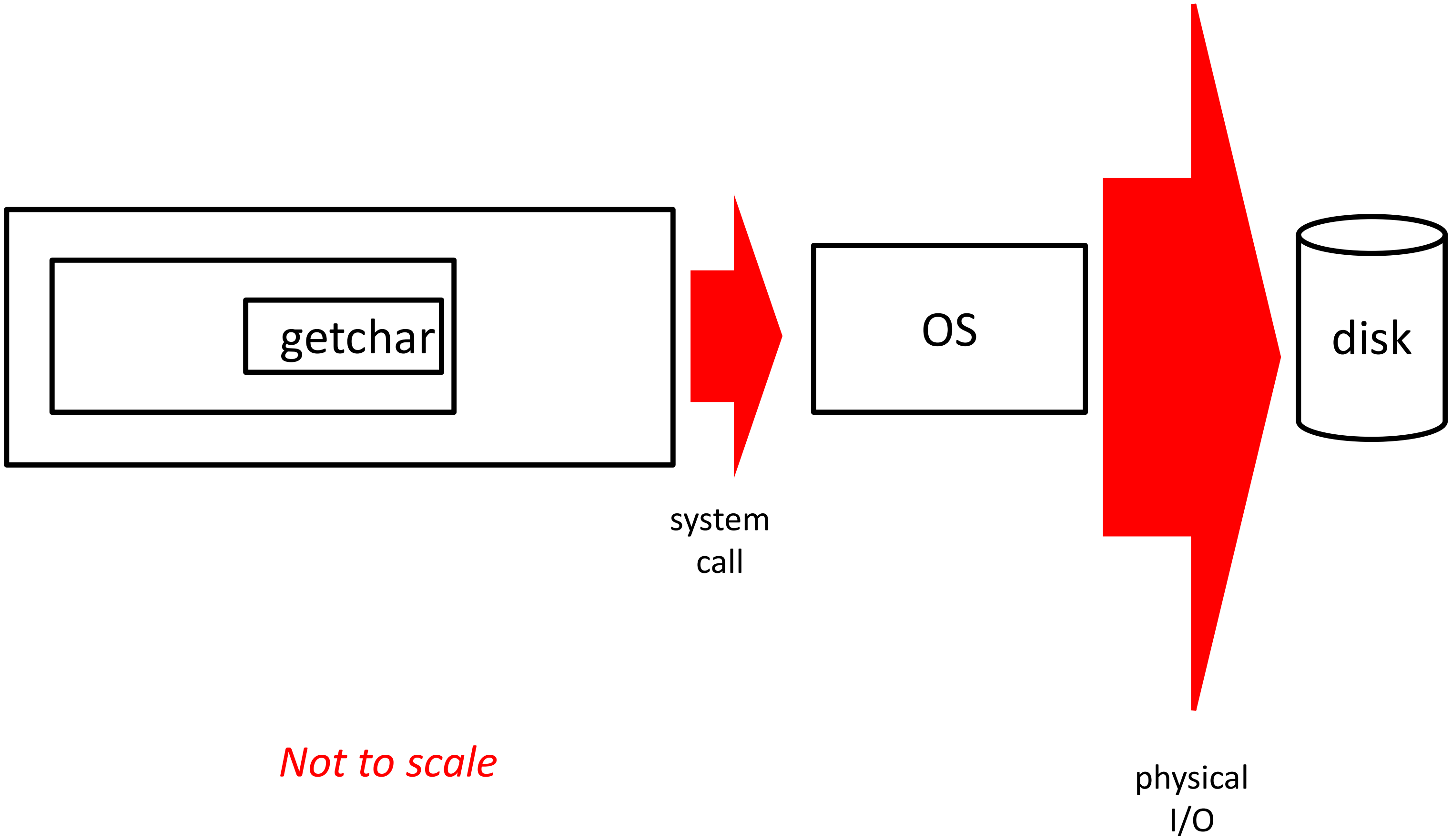
- You might imagine that reading the input character by character is slow



Not to scale

Support for Stream Processing: “inlining”

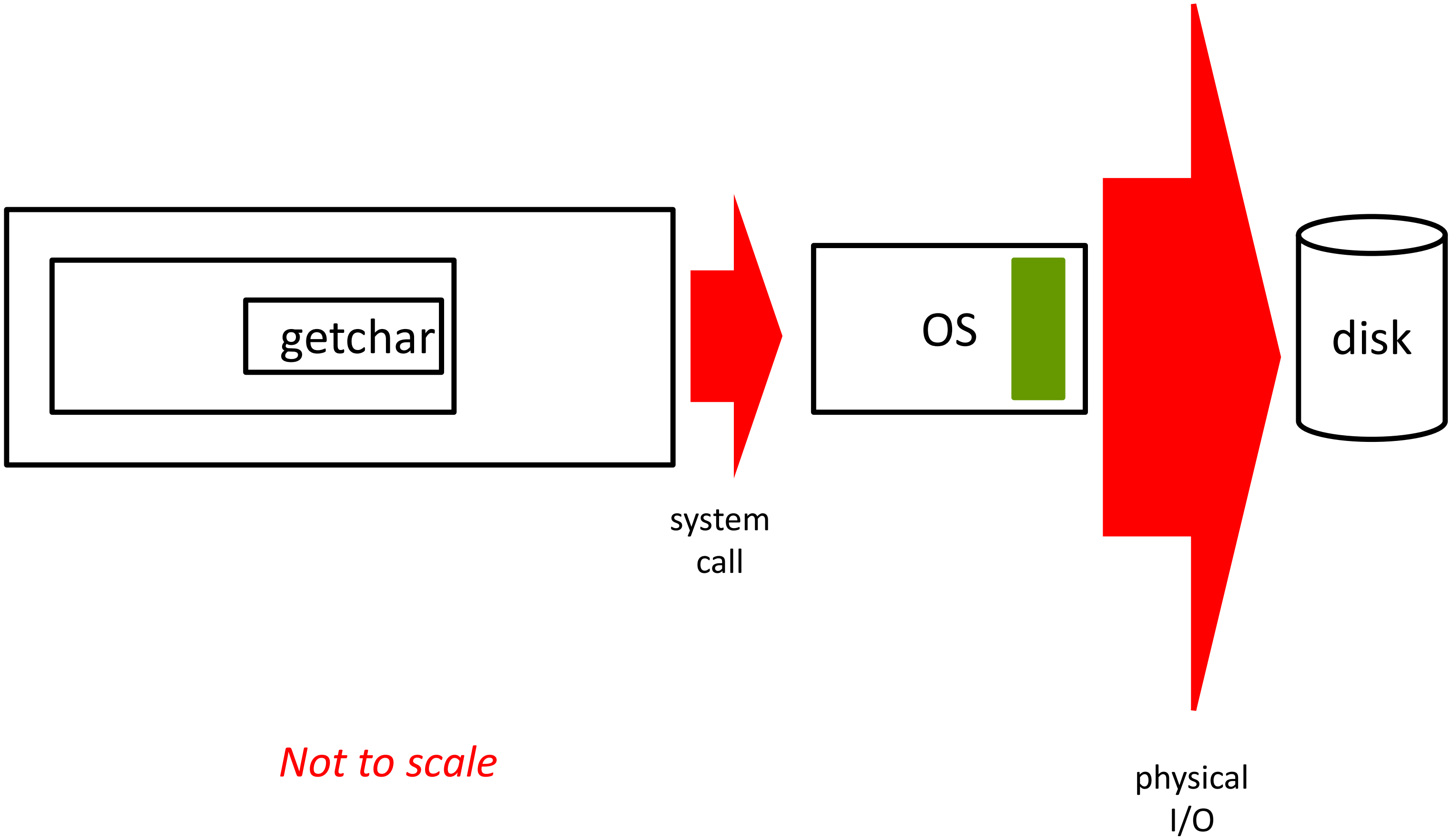
- Some standard library routines are macros



Not to scale

Support for Stream Processing: OS caching

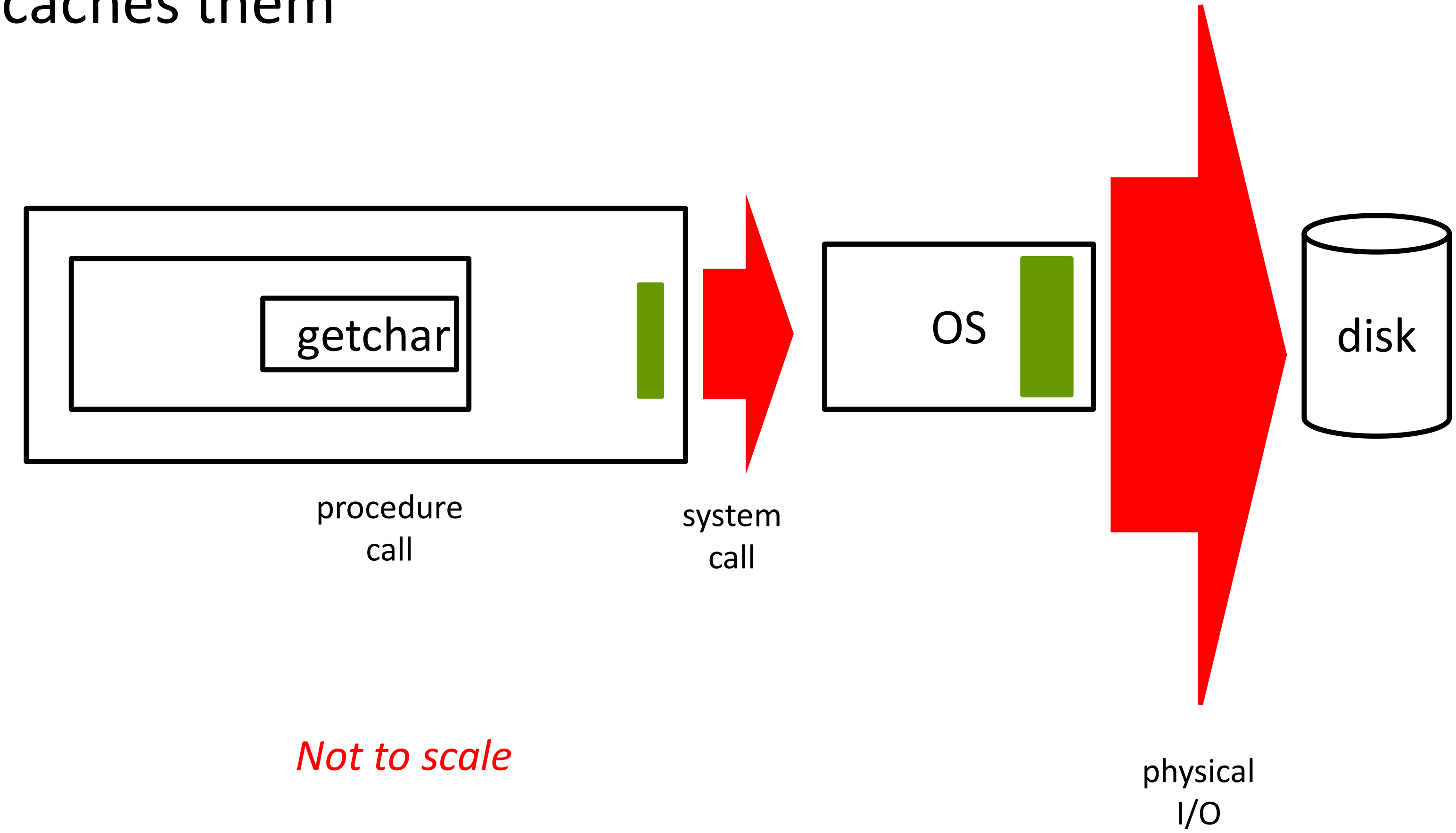
- The OS reads a substantial amount of data and caches it



Not to scale

Support for Stream Processing: libc caching

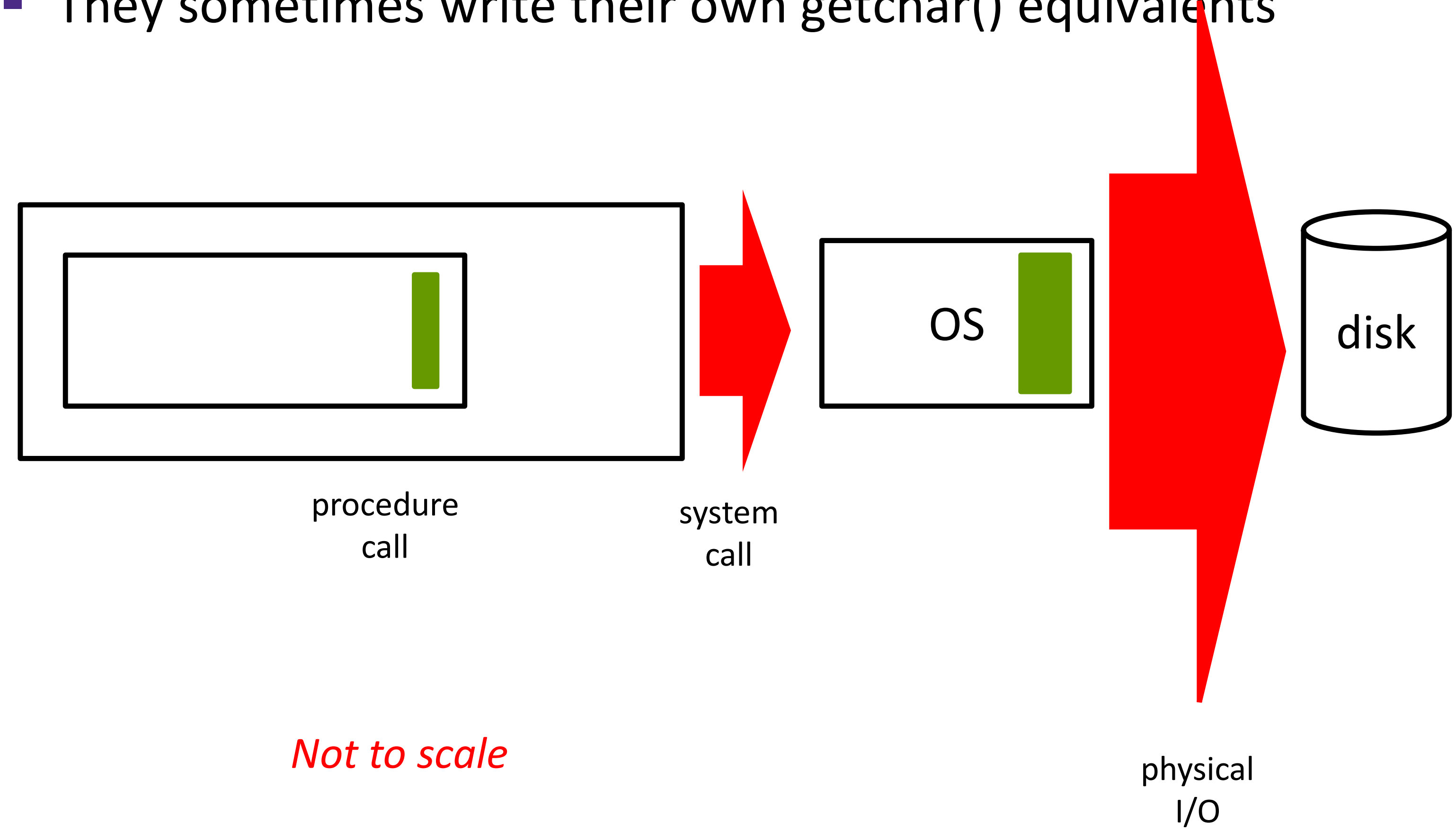
- If you use the FILE* interface, libc reads big pieces and caches them



Not to scale

Support for Stream Processing: app caching

- Some apps read big pieces and cache them
  - They sometimes write their own getchar() equivalents



Not to scale

Support for Stream Processing: C File Interfaces

- C provides 2 file interfaces
- Library interface - <stdio.h>
  - Formatted operations: printf, scanf, fopen, fclose
    - FILE* infile = fopen(“myfile”, “r”);
  - Also unformatted operations: fread, fwrite
  - libc buffers for you
- System call interface
  - int fin = open(“myfile”, O_RDONLY);
  - ssize_t nread = read(fin, buffer, buffer_size);
  - No format conversion, just read/write buffers of bytes

Summary

- Apps from Components
  - Filters
- Stream Processing Structure
- State Machine Implementation Approach
  - ex05 is out
- Efficient Reading of Streams Requires Big Reads and Possibly Caching
  - C's FILE* interfaces do just that
  - C's file handle (int) interfaces don't