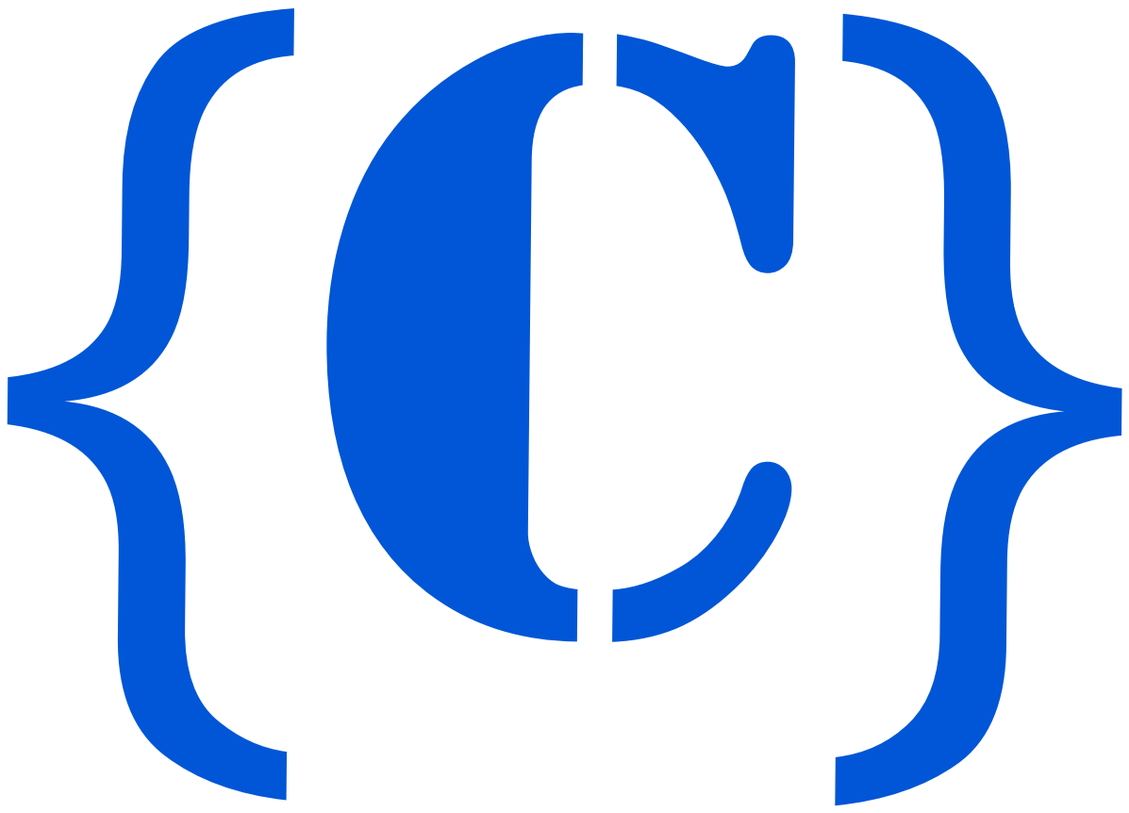


Lecture 17

Callbacks, Asynchronous Programming

Function Pointers

- When used as arguments to functions, they serve multiple roles:
  - They help encapsulate computation that is provided by the caller (e.g., a comparison operator to a sorting routine)
  - They provide a simple form of object-oriented abstraction
  - They enable the callee to update caller's state conditionally, via the behavior of the supplied function

This last feature is commonly referred to as a callback

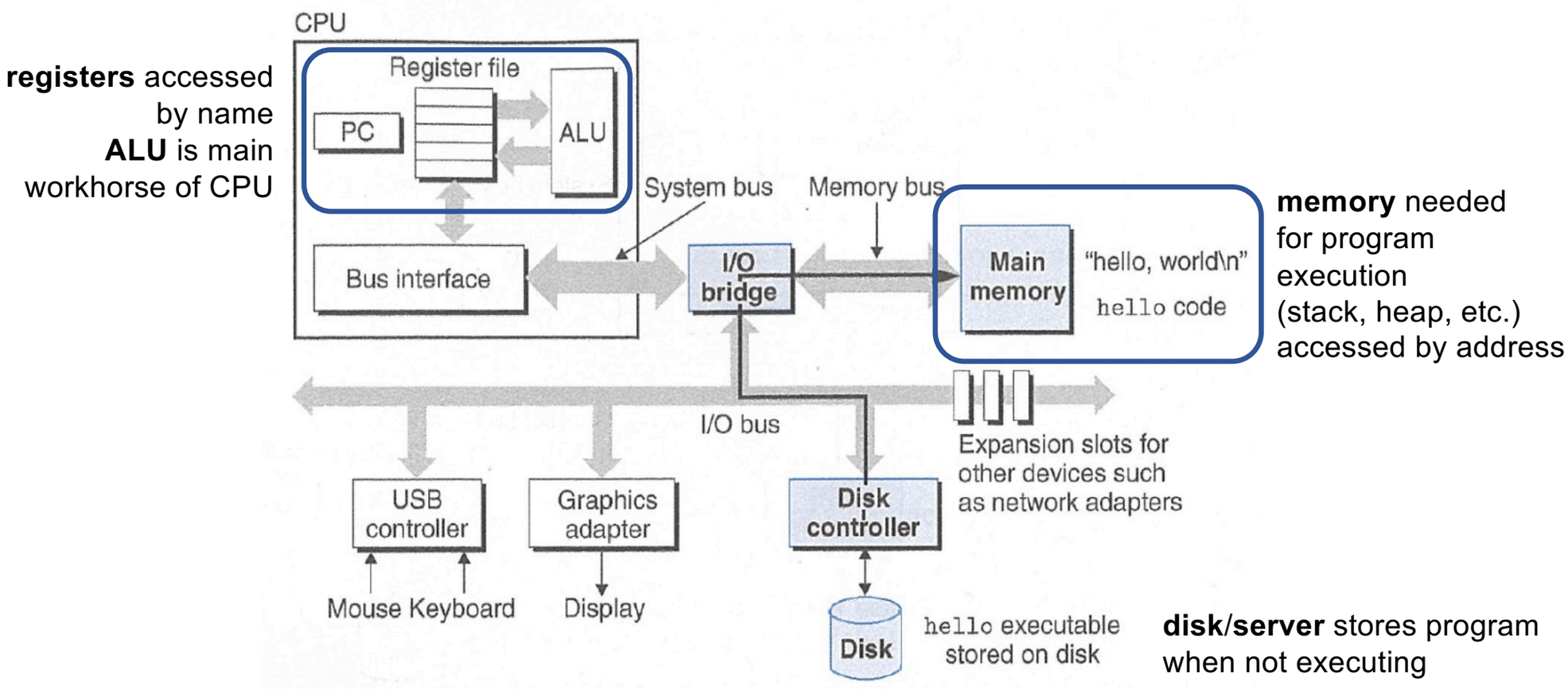
Use Cases

- **Synchronous**: when the callback is invoked, it *fully finishes* its computation before it returns.
  - Purely sequential
  - Example: the comparator operator to Bubble Sort or the function pointer argument to fold or map

- **Asynchronous**: when the callback is invoked, it returns back to the caller *before fully* completing its computation.
  - Implicitly concurrent
  - Example:
    - Network I/O
    - a callback might filter packets sent on a network
    - Once invoked, it returns immediately to allow the program to continue to work
    - It processes packets in the background

Context

- C is a sequential language (with concurrency extensions)
- But, its applications are meant to interface closely with operating system services. These services typically execute concurrently with each other
  - Devices (hard disks, network controllers, GUIs, and peripherals)
  - Processes (User and OS/kernel)
- Callbacks can be used to interact with these services



Example

Asynchronous I/O:

- The I/O operations we've seen so far are also synchronous: once invoked, they don't return until they've fully completed.

- If I/O were asynchronous, it would allow operations to return back to the caller immediately, but continuing to work in the background.

- Callbacks are a natural mechanism to control and manage asynchronous I/O actions

Example

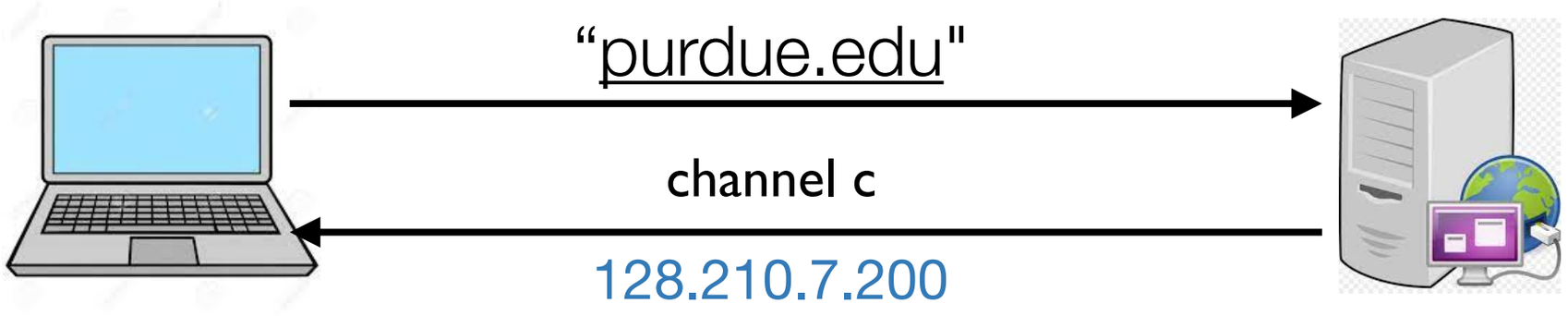
- Consider a name-lookup service (like DNS)
- A client makes a lookup request
- Service returns the IP address of the name

```
void LookupAC(NSChannel_t *c, char *name) {
    int addr;
    SendLookupRequest(c, name);
    RecvLookupResponse(c, &addr);
    printf("Got response %d\n", addr);
}
```

both sending and receiving on the channel is synchronous; caller waits until response is received

receive a response with the address

send a message to the lookup service on a channel



name	address
purdue.edu	128.210.7.200

Example

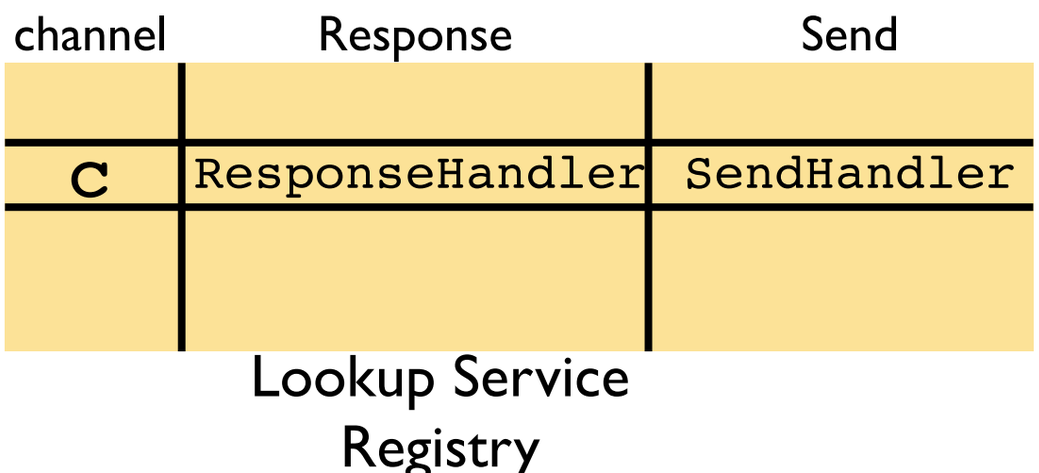
We can use callbacks to allow asynchronous receipt of the response from the server

```
void Lookup(NSChannel_t *c, char *name) {
    OnRecvLookupResponse(c, &ResponseHandler);
    // Store state needed by send handler
    c->st = name;
    OnSend(c, &SendHandler);
  }
  void ResponseHandler(NSChannel_t *c, int addr) {
    printf("Got response %d\n", addr);
}

  void SendHandler(NSChannel_t *c) {
    if (OnSendLookupRequest(c, (char*)(c->st)) == BUSY) {
      OnSend(c, &SendHandler); }}
```

"Register" callbacks

callbacks



When channel is not busy, it calls SendHandler to try to send the message

Generalizing …

- C allows a limited set of interactions with the external environment
- These interactions (and the manner in which they are handled) form C's signal interface
- The signaling mechanism has three parts:
  - The signal name
  - The handler associated with the signal
  - A mechanism to "raise" the signal, i.e., invoke the handler