Programming approaches

Henning Schulzrinne
Dept. of Computer Science
Columbia University

Programming approaches

- data-driven
  - Unix filter model
- event-driven
  - multiple inputs
- web models
  - cgi
  - multi-layer model
- RPC-based models

Data-driven programming

- *transformational*
- input stream ⇒ f(input, commandline) ⇒ output stream
- errors go to stderr
- status: return code
- e.g. *pipe*,
  - sort –f < in.dat | uniq > out.dat
- Advantages:
  - small, modular tools
  - easy to script

Data-driven programming

- Problems:
  - line-oriented output
  - doesn’t work well for networks
  - `sort http://www.census.gov/population` ☺?
  - only for shell, not a GUI abstraction
  - unconditional, not tree

Event-driven programming

- *reactive systems*: inputs not all available in advanced, but instead arrive in endless and perhaps unexpected sequences
- Examples of events:
  - keystrokes and mouse movements
  - network requests (e.g., web)
  - exceptions (connection failed)
  - file input
  - directory or file has changed
  - resource ready (e.g., slow output device)

Event-driven programming

- Asynchronous vs. synchronous:
  - synchronous: wait until operation completes
  - asynchronous: program is notified when operation completes

Events in Unix

- Two event models:
  - signals – one bit
  - select/poll – wait for file system or network events
- Related: condition variables (later)
- Some OS are message-based
- *Handler* or *event loop*

signals

- Software interrupts for *asynchronous* events
- Similar to hardware interrupts
- Provide no information beyond name (integer) – SIGxxx
- Causes:
  - control keys on terminal
  - hardware exceptions:
    - divide by 0
    - invalid memory reference (SIGSEGV),
    - unaligned access (SIGBUS)
  - kill() or kill command
  - software conditions (SIGURG, SIGPIPE, SIGALRM, SIGCHLD)

Signal handling

- Signals can be ignored (most of them) or caught
- Default actions:
  - ignore
  - catch
  - abort
  - abort with core dump

signal()

```
void (*signal(int signo, void(*func)(int)))(int);
```

- sets signal handler for signo to func
- returns previous disposition
- function:
  - SIG_IGN
  - SIG_DFL
- handler returns to calling location, exit() or longjmp()

signal()

```
while (!done) {
  do something
}
void handler(int sig) {
  done = 1;
}
```

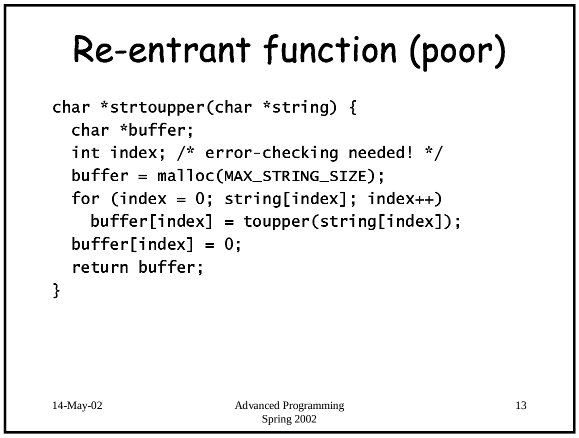
- only call re-entrant functions:

  "A reentrant function does not hold static data over successive calls, nor does it return a pointer to static data. All data is provided by the caller of the function. A reentrant function must not call non-reentrant functions."

Non-re-entrant function

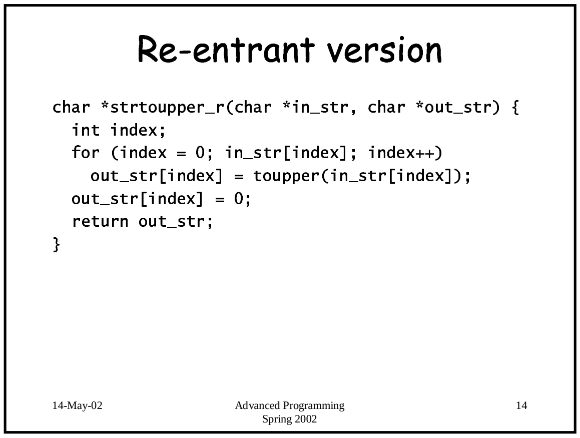
```
char *strtoupper(char *string) {
  static char buffer[MAX_STRING_SIZE];
  int index;
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0;
  return buffer;
}
```

(from AIX manual)



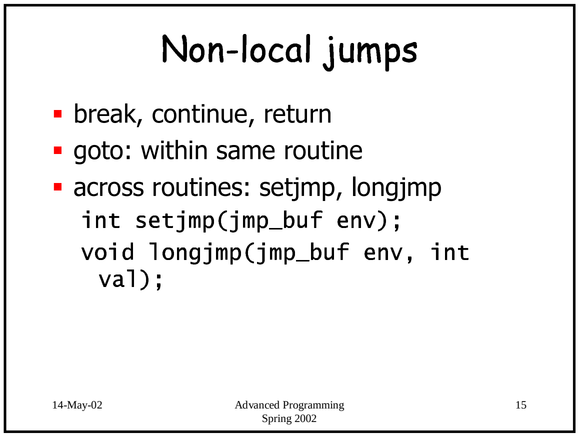
Re-entrant function (poor)

```
char *strtoupper(char *string) {
  char *buffer;
  int index; /* error-checking needed! */
  buffer = malloc(MAX_STRING_SIZE);
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0;
  return buffer;
}
```



Re-entrant version

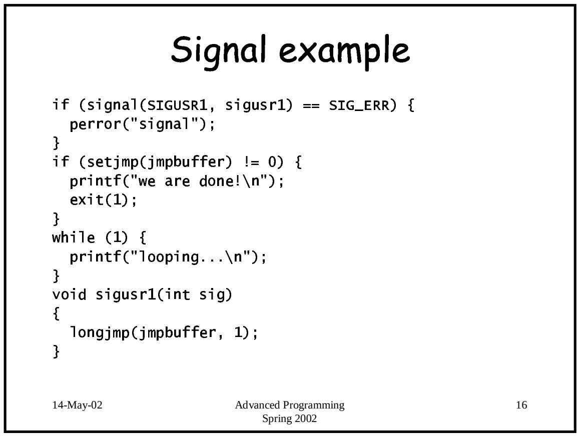
```
char *strtoupper_r(char *in_str, char *out_str) {
  int index;
  for (index = 0; in_str[index]; index++)
    out_str[index] = toupper(in_str[index]);
  out_str[index] = 0;
  return out_str;
}
```



Non-local jumps

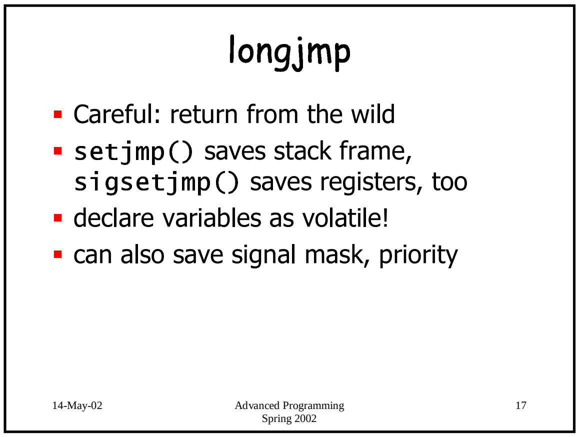
- break, continue, return
- goto: within same routine
- across routines: setjmp, longjmp

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int
  val);
```



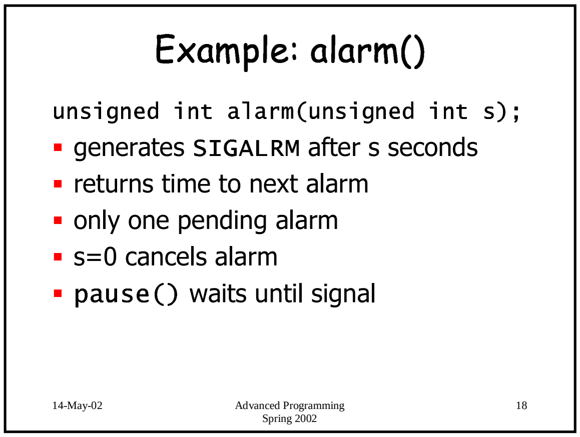
Signal example

```
if (signal(SIGUSR1, sigusr1) == SIG_ERR) {
  perror("signal");
}
if (setjmp(jmpbuffer) != 0) {
  printf("we are done!\n");
  exit(1);
}
while (1) {
  printf("looping...\n");
}
void sigusr1(int sig)
{
  longjmp(jmpbuffer, 1);
}
```



longjmp

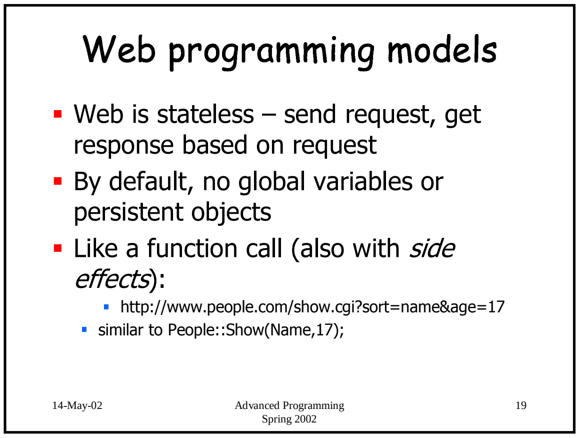
- Careful: return from the wild
- `setjmp()` saves stack frame, `sigsetjmp()` saves registers, too
- declare variables as volatile!
- can also save signal mask, priority



Example: alarm()

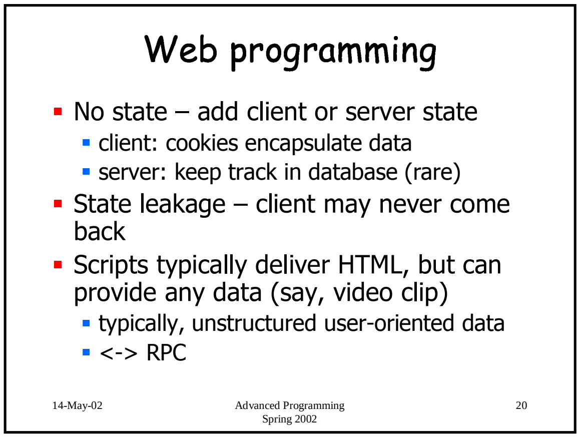
`unsigned int alarm(unsigned int s);`

- generates `SIGALRM` after s seconds
- returns time to next alarm
- only one pending alarm
- s=0 cancels alarm
- `pause()` waits until signal



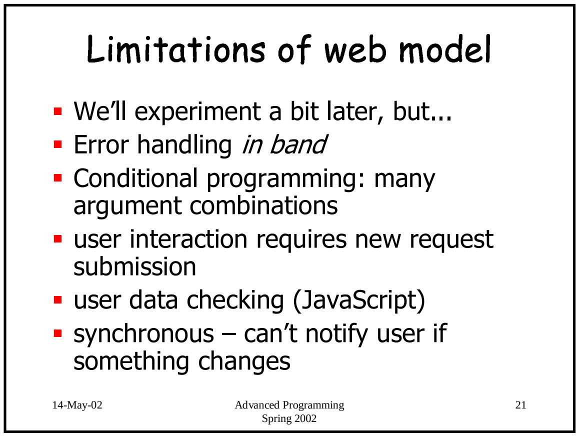
Web programming models

- Web is stateless – send request, get response based on request
- By default, no global variables or persistent objects
- Like a function call (also with *side effects*):
    - http://www.people.com/show.cgi?sort=name&age=17
  - similar to People::Show(Name,17);



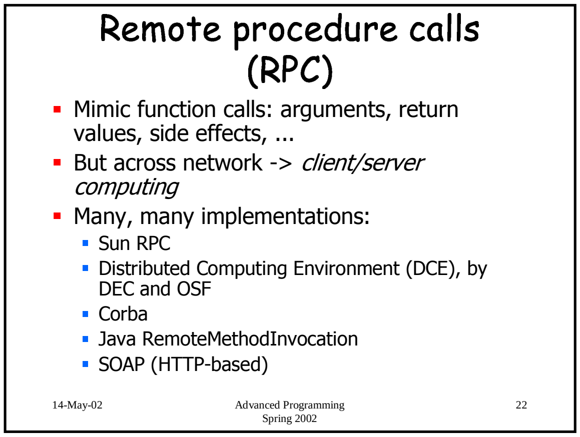
Web programming

- No state – add client or server state
  - client: cookies encapsulate data
  - server: keep track in database (rare)
- State leakage – client may never come back
- Scripts typically deliver HTML, but can provide any data (say, video clip)
  - typically, unstructured user-oriented data
  - <-> RPC



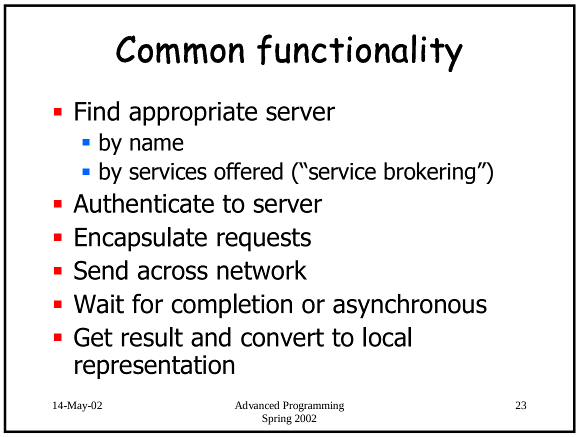
Limitations of web model

- We'll experiment a bit later, but...
- Error handling *in band*
- Conditional programming: many argument combinations
- user interaction requires new request submission
- user data checking (JavaScript)
- synchronous – can't notify user if something changes



Remote procedure calls (RPC)

- Mimic function calls: arguments, return values, side effects, ...
- But across network -> *client/server computing*
- Many, many implementations:
  - Sun RPC
  - Distributed Computing Environment (DCE), by DEC and OSF
  - Corba
  - Java RemoteMethodInvocation
  - SOAP (HTTP-based)



Common functionality

- Find appropriate server
  - by name
  - by services offered ("service brokering")
- Authenticate to server
- Encapsulate requests
- Send across network
- Wait for completion or asynchronous
- Get result and convert to local representation