

Concurrent Architectures: UNIX Part #2

Alistair Rendell

Material drawn from

<http://pages.cs.wisc.edu/~pb/640/sockets.ppt>

<http://www.cs.toronto.edu/~demke/469F.07/index.shtml>

<http://www.cs.cf.ac.uk/Dave/C>

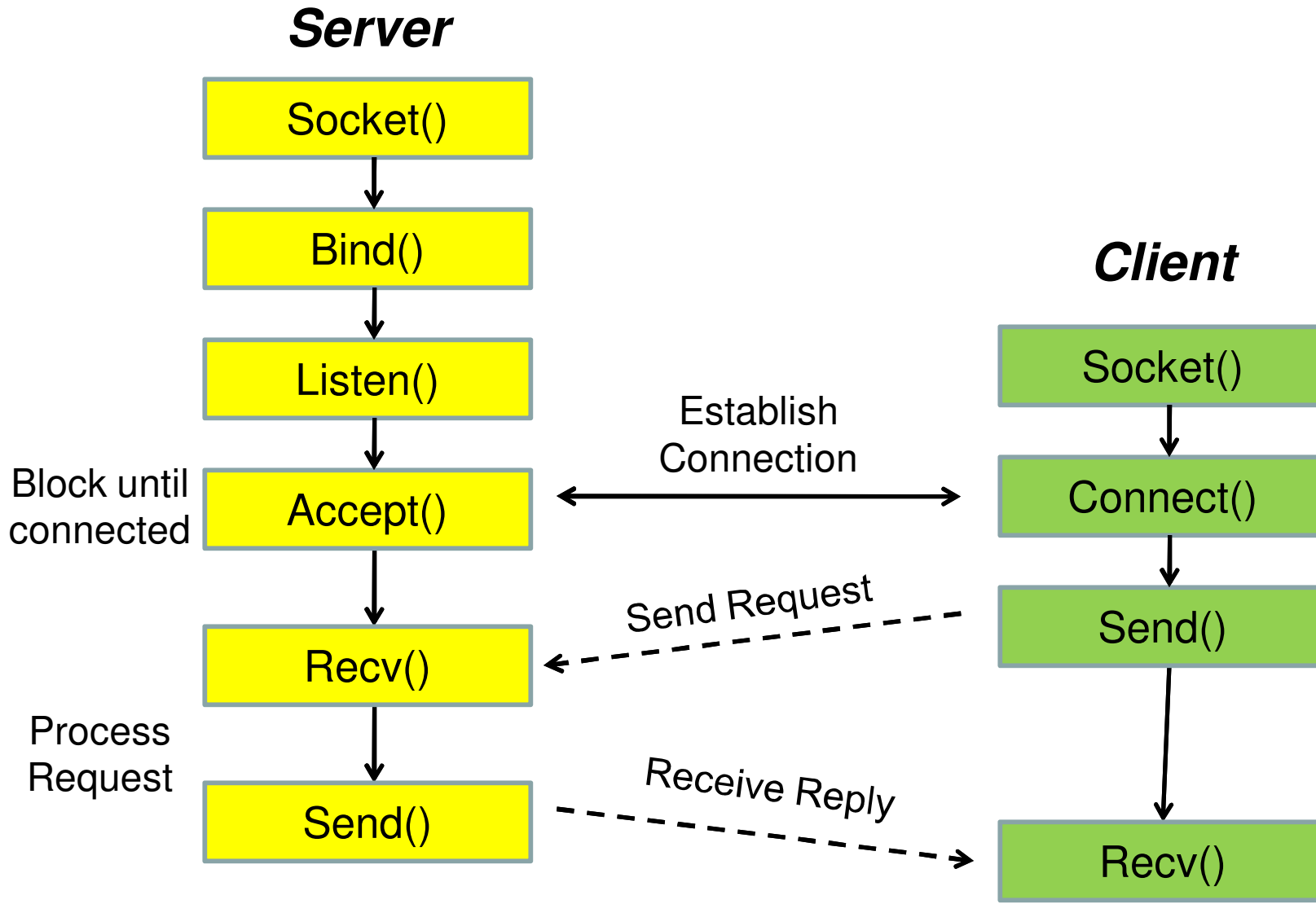
Support for Concurrency in UNIX

- Creating processes
 - `fork` (and `exec`)
- Communicating between processes
 - `pipe` (lab 7)
- Enabling communication between computers
 - `socket`
- Handling non-deterministic events
 - `select` (lab 8)
 - Interrupts/signals (lab 9)
- Some other stuff
 - Semaphores (see lab 9)
 - Message queues (not covered in detail)
 - Shared memory segments (not covered in detail)

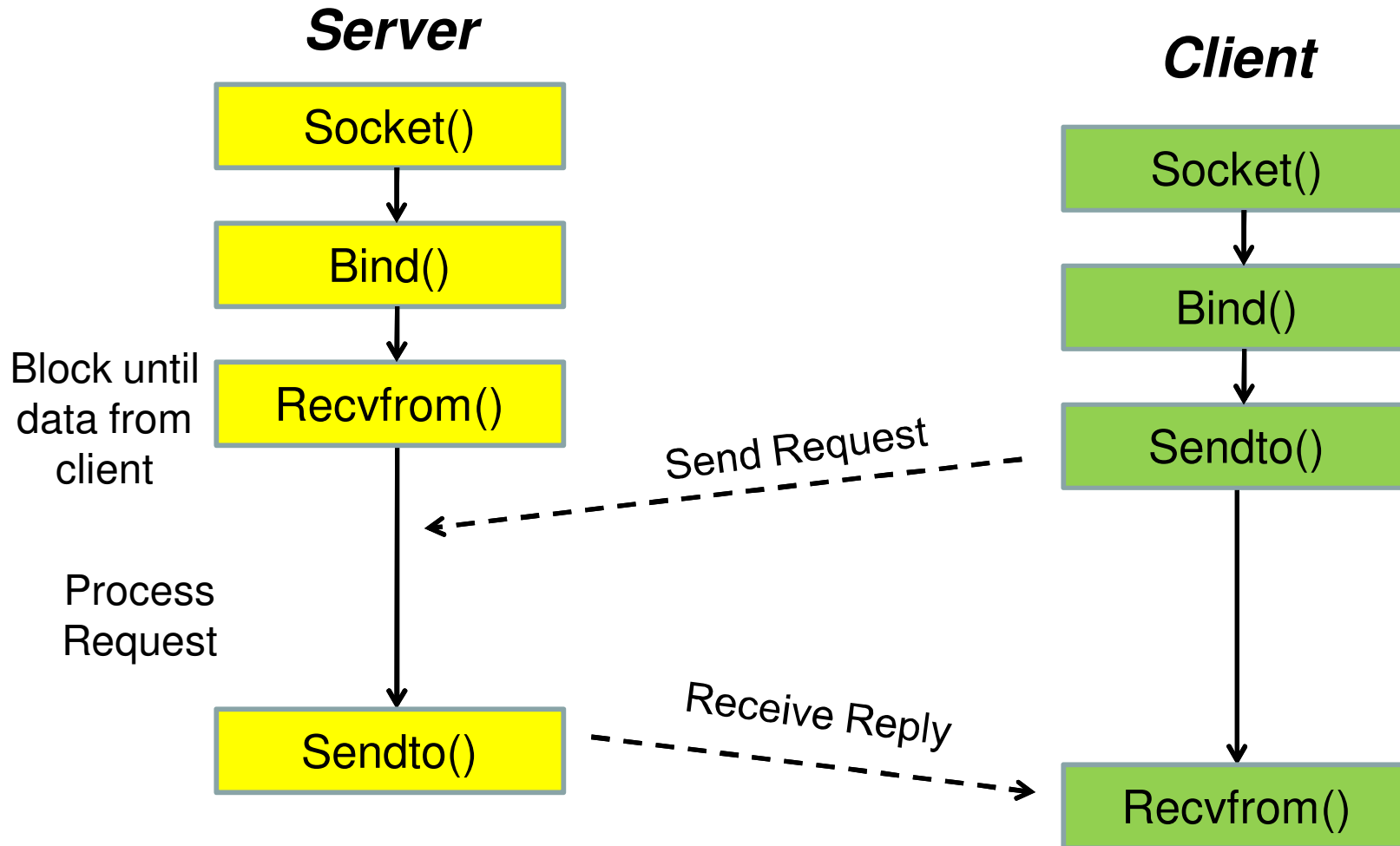
Berkeley Sockets

- Networking protocol originally provided by BSD 4.1c ~ 1982
- The networking API exported by most OS's
- Principle abstraction is a socket
 - Where the application attaches to the network
 - Define operations for creating connections, attaching to network, sending/receiving data and closing connection
- Two forms
 - Connection oriented, Transmission Control Protocol (TCP)
 - Connectionless: User Datagram Protocol (UDP)

Connection-Oriented (TCP)



Connectionless (UDP)



The Socket Call

- Means by which application attaches to the network

```
int socket(int family, int type, int protocol)
```
- Family: address family (protocol family)
 - AF_UNIX, AF_INET, AF_NS, AF_IMPLINK
- Type: semantics of communication
 - SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
 - not all combinations allowed
- Protocol: usually set to 0 but can be set to specific value
 - Family and type usually implied by the protocol
- Return value is a handle for (a file descriptor, fd) for new socket

The Bind Call

- Binds a newly created socket to the specified address

```
int bind(int socket, struct sockaddr
        *address, int addr_len)
```

- `socket`: newly created socket handle (fd)
- `address`: data structure of address of local system
 - IP address and port number
 - same operation for both connection-oriented and connectionless servers
 - can use well known port or unique port
 - bunch of routines to convert different forms of address representation (see lab)

The Listen Call (server side)

- Used by connection-oriented servers to indicate an application is willing to receive connections

```
int listen(int socket, int backlog)
```

- `socket`: handle (fd) or newly created socket
- `backlog`: number of connection requests than can be queued by the system while waiting for server to execute accept call

The Accept Call (server side)

- After execution of listen, the accept call carries out a passive open (server prepared to accept connects)

```
int accept(int socket, struct sockaddr
           *address, int addr_len)
```

- Blocks until a remote client carries out a connection request
- Returns with a new socket for the new connection where the address contains the clients address

The Connect Call (client side)

- Client executes an **active open** of a connection

```
int connect(in socket, struct sockaddr
            *address, int addr_len)
```

- Call does not return until the three-way handshake (TCP) is complete (or an error)
- Address field contains remotes system's address
- Client OS usually selects random unused port)

Send(to), Recv(from)

- After connection has been made, application uses send/recv to transfer data

```
int send(int socket, char *message, int  
msg_len, int flags)
```

- Sends specified message using specified socket

```
int recv(int socket, char *buffer, int  
buf_len, int flags)
```

- Receives message from specified socket into specified buffer

Socket Advice

- Read man pages very carefully for systems you are using
- Use a separate process to implement each protocol
- Applications use buffers as do protocols, copies are expensive, message abstraction aims to minimize this
- Do not use ports <1024 , these are reserved for root process
- client and server can be on same system
- Tools and info include
 - netstat, ifconfig, ping, traceroute, /proc, tcpdump etc

Non-Determinism: Select

- Unix provides the select system call to block on more than one source at a time, waking up when there is information from any of them.
 - select actually allows you to wait at the same time for messages, events, or write operations

But compared to Ada select UNIX select is very messy!

UNIX Select

```
#include <sys/types.h>
#include <sys/time.h>
int select( int nfd,          /* Highest number fd in any set + 1 */
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout); /* null means until activity*/

FD_ZERO(&fdset)      /* fdset becomes the empty set */
FD_SET(fd,&fdset)     /* fd is added to fdset */
FD_CLR(fd,&fdset)     /* fd is removed from fdset */
FD_ISSET(fd,&fdset)  /* it determines if fd is in fdset */
```

- Upon return from select readfds, writefds, exceptfds have set only the bits that correspond to ready files.
- The select function returns the number of bits that are set in readfds, writefds, exceptfds.
- Use FD_ISSET to determine which fd had activity on them.

Example of usage in Lab 8

Non-Determinism: Interrupts and Signals

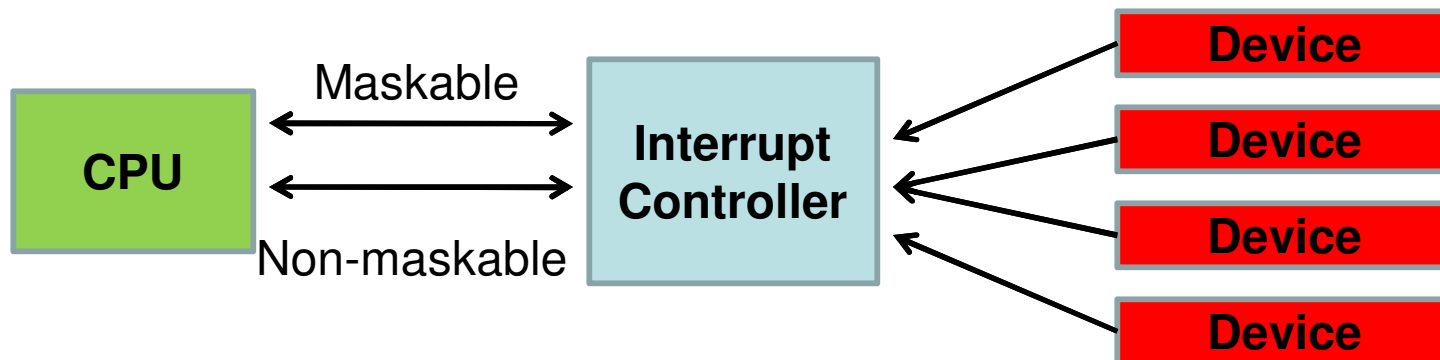
- Definition:
 - An event external to the currently executing process that causes a change in the normal flow of instruction execution (usually generated by hardware devices external to the CPU)
- Key issue is that interrupts are asynchronous to current process
- Typically implies that some device needs service

Why Interrupts

- To allow multiple devices to be connected
- Devices need CPU service but can't predict when
- Eternal events typically happen on a different timescale compared to CPU
 - want to keep CPU busy between events
- Need a way for CPU to find out a device needs attention
- Solutions
 - Polling (why is this bad, when is it good?)
 - Interrupts

Interrupts

- Give each device a wire (interrupt line) that it can use to signal the processor
 - When interrupted the processor executes a routine called an interrupt handler
 - No overhead when no request pending



Hardware Interrupt Handling

- Details are architecture dependent
- Interrupt controller signals CPU that interrupt has occurred, passes interrupt number
 - interrupts are assigned priorities to handle simultaneous interrupts
 - lower priority interrupts may be disabled during service
- CPU senses (checks) interrupt request line after every instruction:
 - if raised uses interrupt number to determine which handler to start
- Basic state saved (as for system call)
- CPU jumps to interrupt handler
- When interrupt done, program state is reloaded and program resumes

Software Interrupt Handling

- Typically two parts to interrupt handling
 - What must be done immediately
 - so the device can keep running
 - What can be deferred until later
 - so that response is faster
 - so we can have a more convenient execution context...

Interrupt Context

- Execution of the first part of interrupt handler “borrows” the context from whatever was interrupted
 - interrupted process state is saved in process structure
 - handler uses interrupted threads kernel stack
 - handler is not allowed to block
 - no process structure of its own to save state or allow rescheduling)
 - can't call functions that might block
- Handler needs to be kept as fast and simple as possible
 - typically sets up work for second part, flags that second part needs to be executed, and re-enables interrupt

Software Interrupts

- The deferred parts of interrupt handling are sometimes referred to as “software interrupts”
- Networking example
 - time critical: copy packet off hardware
 - deferred work: process packet, pass to correct application
- Timers example
 - Time-critical: increment current time-of-day
 - Deferred: recalculate process priorities

Signals

- Software equivalent of hardware interrupts
- Allows process to respond to asynchronous external events
 - process may specify its own signal handlers or may use OS default action
 - Defaults include
 - ignore the signal
 - terminate all threads in process (with or without core dump)
 - stop all threads in a process
 - resume all threads in a process
- Signals provide a simple form of inter-process communication

Basics

- Process structure has flags for possible signals and actions to take
- When signal is posted to process, signal pending flag is marked
- When process is next scheduled to run, pending signals are checked and appropriate action take
 - signal delivery is not instantaneous

Signal Terminology

- Posting
 - action taken when event occurs that process needs to be notified of (aka signal generation)
- Delivery
 - action taken when process recognizes arrival of event (aka signal handling)
- Catching
 - if user-level signal handler is invoked, process is said to catch the signal
- Pending
 - signals that have been posted but no yet delivered

User-Level View

- Write a signal handler function
 - eg to handle SIGINT (usually ctrl C from keyboard)

```
void signint_handler(int sig) {  
    printf("Interrupted!\n");  
    fflush(stdout);  
}
```

- Install it

```
struct sigaction new_action, old_action;  
new_action.sa_handler = signint_handler;  
sigaction(SIGINT, &new_action, &old_action);
```

Other user-level actions

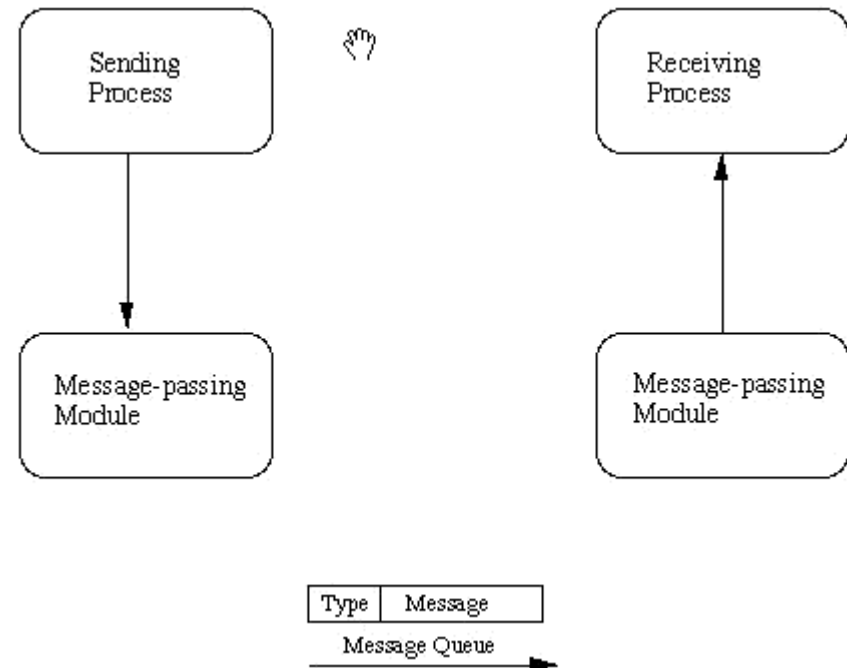
- Block signal delivery by masking signals
 - similar to disabling interrupts
 - `sigsetmask()`
- Specify that signal handlers run on separate stack (useful if signal caused by stack overflow)
 - `sigaltstack()`
- Retrieve list of pending signals
 - `signpending()`
- Block process until signal is posted
 - `sigsuspend()`
- Send signal to process
 - `kill()`

Complications

- Handler may execute at any time
 - need to be careful of manipulating global state in a signal handler
- Signal delivery may interrupt execution of signal handler
 - code should be re-entrant
 - need to block signals
- Only one signal handler per signal per process
 - can't use in library code
- Usually no signal queuing

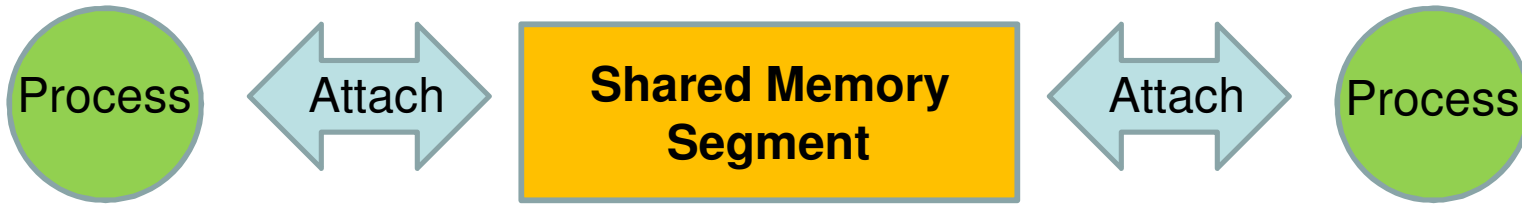
Other Stuff: Message Queues

- Two (or more) processes can exchange information via access to a common system message queue.
 - The *sending* process places via some (OS) message-passing module a message onto a queue which can be read by another process
 - Each message is given an identification or type so that processes can select the appropriate message.
 - Process must share a common key in order to gain access to the queue in the first place
- Message queues persist after end of process that creates them



See <http://www.cs.cf.ac.uk/Dave/C/>

Shared Memory Segments



- A process creates a shared memory segment using `shmget()`
 - the original owner of a shared memory segment can assign (or revoke) ownership to another user with `shmctl()`.
- Once created, a shared segment can be attached to a process address space using `shmat()`.
 - It can be detached using `shmdt()` (see `shmop()`).
 - The attaching process must have the appropriate permissions.
- Once attached, the process can perform normal read or write operations to the segment, as allowed by the permissions.
- A shared segment can be attached multiple times by the same process.
- Shared memory segments persist after end of process