Processes, Threads, and Process States

Programs and Processes

- Program: an executable file (before/after compilation)
- Process: an instance of a program that is active in the system
  - A program can be written to activate new processes

- Two important issues about processes
  - Process Control Block (PCB)
  - Process states
    - During execution, the process can be in one of the several states
    - Ready, running, blocked, etc.

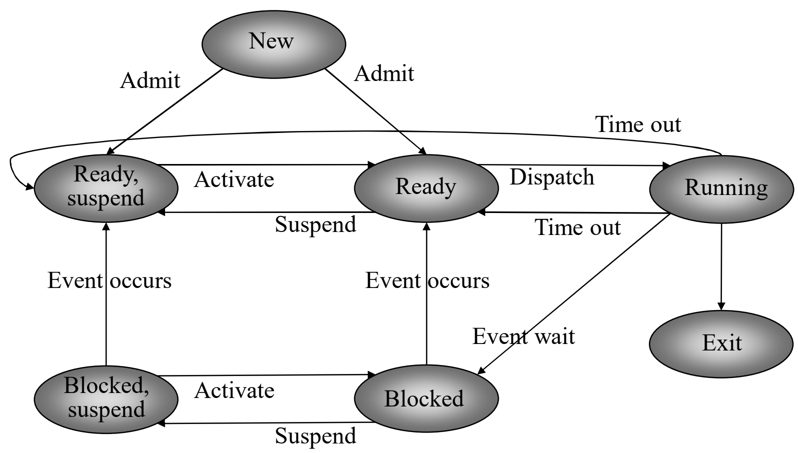
Processes Control Block (PCB)

- PCB: A data structure used by the OS to keep track of the information related to a process
- What information is needed?
  - About the process itself
    - Process id
    - Execution priority
    - Accounting: resource usage (like how much CPU time used, etc.)
- What else?
  - How about context switch
    - Need space to store contents of the registers
    - The information is in PCB
- What else?

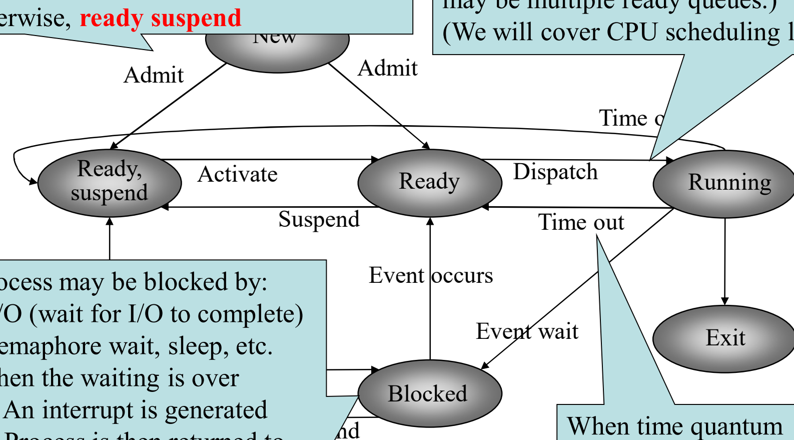
Processes Control Block (PCB)

- What else?
  - To execute a program, what is being used?
  - Memory, I/O devices, files
  - PCB keeps pointers to
    - Memory segments (pages) in use
    - Files opened
      - File descriptors
    - I/O information
      - Sockets opened
      - Other I/O device such as monitor, keyboard, etc.

Process States



During execution, a process can be in one of the 7 states (7 state model)



A new process is created ("run" program)
If the system has sufficient memory, then the new process is loaded into memory and placed in **ready queue**
Otherwise, **ready suspend**

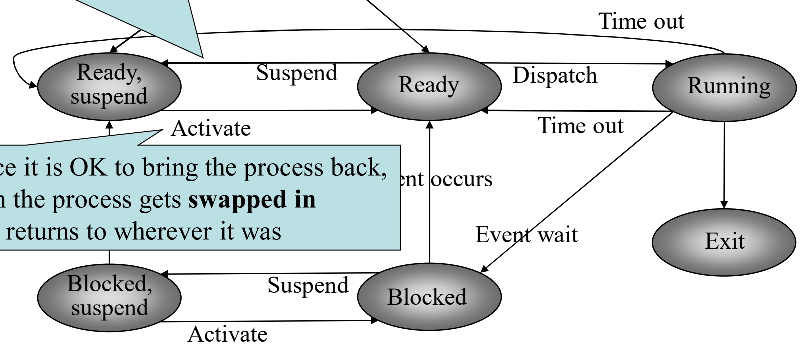
Dispatch: CPU scheduler takes a process from the head of the ready queue to execute (sometimes, there may be multiple ready queues.)
(We will cover CPU scheduling later.)

Process may be blocked by:
- I/O (wait for I/O to complete)
- semaphore wait, sleep, etc.

When the waiting is over
→ An interrupt is generated
→ Process is then returned to ready queue

When time quantum expires,
the process is returned to ready queue

When the system runs low on memory,
a process may be **swapped out** and
goes into the **suspend state**
- If from blocked state, then blocked suspend
- If from ready state, then ready suspend



Once it is OK to bring the process back,
then the process gets **swapped in**
and returns to wherever it was

During execution, a process can be in one of the 7 states (7 state model)

Each state has a queue to keep track of the processes in that state

Kernel Mode

- Kernel mode and user mode
  - In running state, a process may run in user/kernel mode
    - Regularly: processes run in user mode
  - When running some privileged instructions: kernel mode
    - Systems need to prevent users from doing something that may damage the system, e.g., destroy the file system or simply need to force the access control of files or memory frames
    - $\Rightarrow$ Some instructions becomes privileged $\Rightarrow$ Hardware will stop executing privilege instructions unless the running mode is kernel
  - How a process gets to run in kernel mode
    - Only via system calls that specifically call a “trap” instruction
    - After trap, the code to execute can only be system code that has been carefully inspected

Process State Transition

- When process state changes
  - The PCB (ptr) will be moved to the associated queues
- Process switch
  - Some state transitions result in process switch
    - Running ⇒ ready / blocked / suspended
    - E.g., time quantum expiration, process gets blocked
    - Current running process is switched out and the next process in the queue is switched in
- Consider process X:
  - load a -- load a to "AC"
  - add b -- add b to "AC"

X is switched out in between.
What is AC when X comes back?

Context Switch

- Process has context
  - = State information of the process
  - Needs to be saved when gets switched out
  - Needs to be restored when switched in
- What should be saved?
  - Memory content? Disk content?
  - Registers in the CPU?
  - Which registers?
    - PC, IR, PSW, stack pointer, MAR, MBR?

Processes and Threads

- A process is a unit of
  - Resource ownership (address space, I/O devices, files, etc.)
    - E.g., each process has its own address space
  - Dispatching (process state)
- Thread is a only a unit of
  - Dispatching
  - Not for resource ownership
    - It is a light-weight process
- A process may create several processes
  - Each of them has its own address space
- A process may create several threads
  - All of them share the same address space

Create a Process

- Users can create processes also, but how?
- In Unix: fork () -- create a child process

```
main()
{ int pid;
  printf ("before fork\n");
  pid = fork();
  if (pid == 0)
        printf (“child process");
  else
        printf (“parent process");
}
```

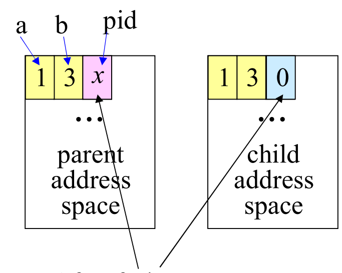
Each process has a process id
- use “ps -al” to see process id
- Or use getpid() in the code
- Parent process gets the child pid
  in the return value from fork
- Child process gets a pid
  but its fork return value is 0

Processes

❖ Unix process creation

```
main()
  { int pid;

    a=1; b=3;
    printf ("before fork");
    pid = fork();
    if (pid == 0)
       printf ("child process");
    else
       printf ("parent process");
  }
```



After fork
- parent gets **child process ID**
  (needed if parent needs to wait for a specific child to finish)
- child process gets **pid = 0**

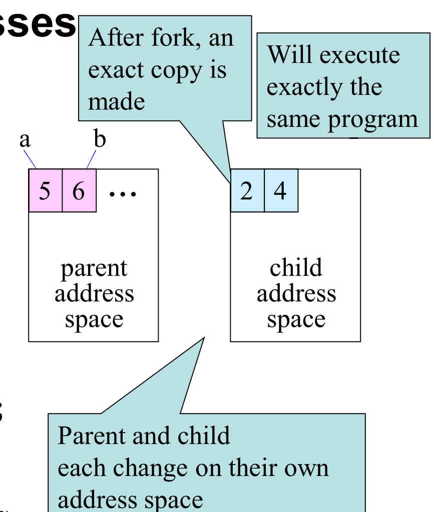
Note: pid can be any variable
```
p = fork();
if (p == 0) printf ("child");
else printf ("parent");
```

Processes

❖ Unix process creation

```
main()
  { int pid;

    a=1; b=3;
    printf ("before fork");
    pid = fork();
    if (pid == 0)
       a=2; b=4;
       printf ("child process");
    else
       a=5; b=6;
       printf ("parent process");
  }
```



Processes

❖ Unix process creation

```
main()
  { int pid1, pid2;
    printf ("before fork");
    pid1 = fork();
    pid2 = fork();
    printf ("What happens?");
  }
```

Processes

❖ Unix process creation

```
main()
  { int pid;
    x = 1;
    printf ("before fork");
    pid = fork();
    if (pid == 0)
    { printf ("will x be passed to "new" after exec?");
      exec ("new") -- address space replaced
      printf ("Will this statement be executed?");
    }
    else printf ("parent process");
  }
```

How Processes Communicate

- Unix pipe (example)
  - System offers a shared space
    - User space cannot be shared
  - Pipe creates an entry in the shared space
  - Write: a buffer is allocated to store the message
  - Read: retrieve the message in the pipe
  - Each direction of communication requires a pipe
- Socket (tutorial)
  - Similar to pipe, but allow network communications
  - An I/O descriptor is assigned in the user space, keep track of activities on the network device controller

Pipe for Interprocess Communication

- Example for pipe creation

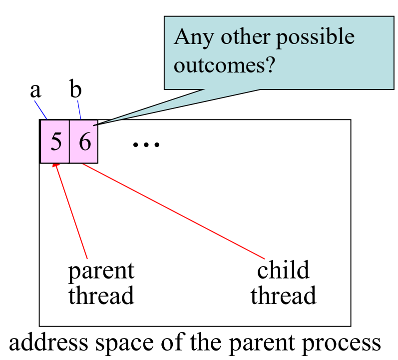
```
int fds1[2], fds2[2];
     // each pipe returns 2 file descriptors, [0] for read, [1] for write
pipe (fds1);   pipe (fds2);
childr = fds1[0];   childw = fds2[1];
parentr = fds2[0];   parentw = fds1[1];
if (fork() > 0)
  { // in the parent process
    // write to parentw and read from parentr
   close (childr); close (childw);
  }
  else
  { // in the child process
    // write to childw and read from childr
    close (parentr); close (parentw);
  }
```

Threads

- A thread is a unit for dispatching
  - Need to have a thread control block (TCB)
- What is needed in TCB
  - Thread id, thread privilege information
  - Execution stack
  - Pointer to the tables of the parent process
- Creation overhead comparison
  - fork(): 1700 *u*sec
  - thread: 52 μsec (user level thread)
  - Much cheaper to create a thread or perform thread switch
  - (Old data based on Sparc-2)

Threads

- A thread does not have its own address space
- Threads created by one process share the address space of the process



Parent thread:

a = 5; b = 6;

…

Child thread:

a = 2; b = 4;

…

Thread Creation

- pthread in Unix
  - E.g., **ret = pthread_create (&tid, NULL, add, NULL);**
  - Pthread_create parameters

    typedef unsigned long int pthread_t;

    - pthread_t *tid : returned value, defined in <sys/type.h>, different versions of Linux have different definitions, but the value is unique
    - Pthread_attr_t *attr : user can set the thread attributes
      - E.g., detachstate: whether the thread is detached;
      - E.g., synch: whether the thread is synchronous
      - NULL ⇒ default attribute values are used
    - Function to be executed by the created thread
      - (void *) (* fun-name) (void * arg)

        (void *) is the return type
        * fun-name passes the function addr
    - void *arg : the argument for the thread function
    - C++ function pointer is not compatible with pthread_create() ⇒ Define a C++ function outside & declare it as: extern "C" function

Thread Implementation

- What should be provided to the user
  - Mechanisms for thread creation and disposal
  - Mechanisms for handling sharing
    - E.g., semaphors or monitors to synchronize memory accesses
- Thread library could be implemented at the
  - User level (not visible to OS)
  - Kernel level (visible to OS)
- Advantages of user-level thread library
  - Lower cost
    - No kernel trap required for thread switch
    - Still need context switch
  - Flexible, customized thread scheduling

Thread Implementation

- Disadvantages of user-level thread library
  - When a thread is blocked (e.g., read, sleep), the entire program is blocked
    - Another thread could execute and make use of the time quantum
    - Solutions:
      - A blocking system call can be wrapped in the thread library and actually a nonblocking call is used (called jacketing); e.g., read becomes nonblocking read, sleep does not really sleep
      - The thread should not be scheduled to run till the condition is satisfied (thread library should check the condition and if not satisfied, do not switch execution to the blocked thread, e.g., check whether read data is ready, check whether sleep time is up)
      - When threads block often, this solution can be more expensive (frequent check of the condition is necessary)

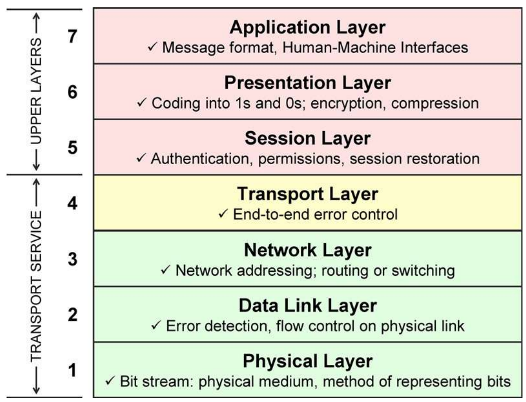
Thread Implementation

- Disadvantages of user-level thread library
  - Cannot make use of multi-core systems
    - Multiple threads in one process will only be scheduled to one core because they are not visible to the kernel

- Inter-thread communication?

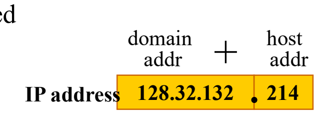
Interprocess Communication via Socket

- Interprocess communication
  - Could be for intra or inter hosts
  - Among processes
- 7-layer network
  - Defined by ISO
- Socket
  - Network programming interface



Interprocess Communication via Socket

- Basic terminologies
  - Packets: header + payload
    - Header: source/destination addresses, protocol used, etc.
  - Protocols
    - When two hosts communicate, they need to understand each other
    - Many protocol families have been defined
      - E.g., TCP, UDP, IP protocols
  - Address



    - How to identify the process you want to communicate with and where it is ⇒ Give addresses to hosts and processes on each host
    - Different protocol families have different addressing mechanisms
    - TCP/IP family: IP address + port number
      - IP address identifies a host
      - port number identifies a specific process (and a specific port)

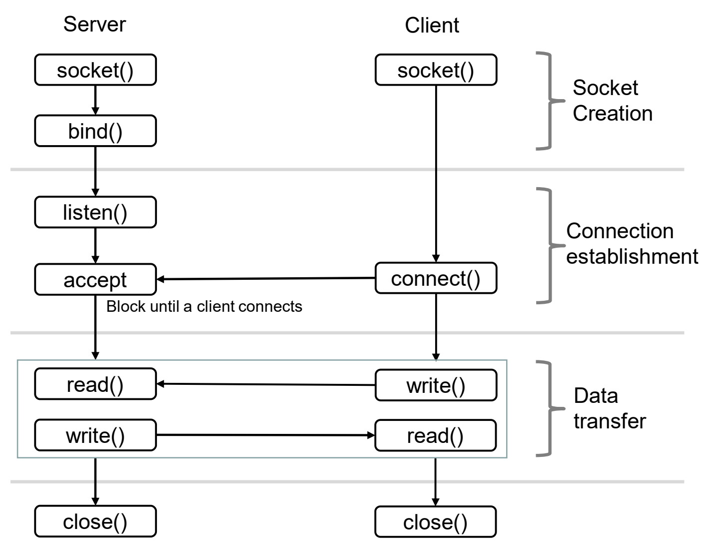
Interprocess Communication via Socket

- Protocols
  - UDP = datagram protocol, connectionless and unreliable
    - No ACK, no retransmission, delivery may be out of order, may have lost or duplicated packets
    - Connectionless, user needs to provide address for each packet
  - TCP = byte-stream protocol, fixed connection, reliable
    - Requires ACK to detect problems, lost packets get retransmitted
    - Once the connection is established, can send/recv multiple times till the connection is closed
  - IP: raw packets
  - Other protocols
    - http, ftp, … (http, ftp are above TCP)
    - Bluetooth, wifi, GSM, LTE (different protocol families from IP's)

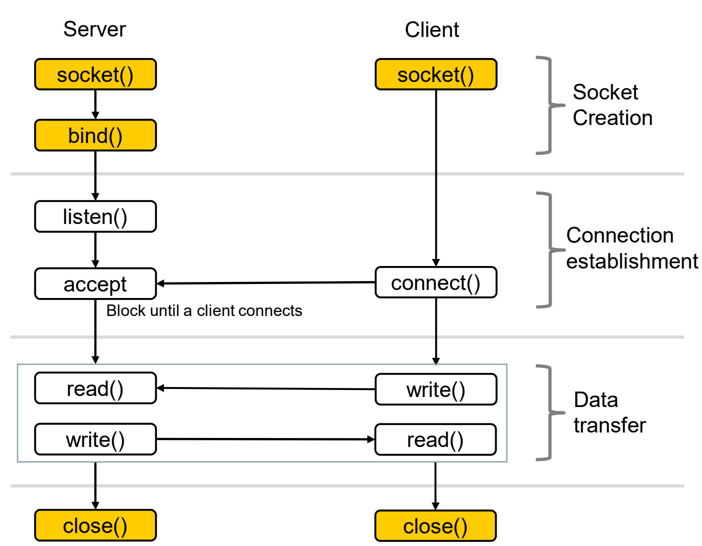
Interprocess Communication via Socket

- Berkley sockets
  - Before the **socket** package was created, OS venders offer network programming packages, many different versions
    - They follow IETF protocols and can communicate with each other
    - But programmers need to know vender specific packages
  - Berkley created BSD **socket** API
    - Released in 1983, with 4.2 BSD Unix
    - Offer generic access to network communications
  - Subsequently, all OS supports socket API
    - Become the standard network programming API

Socket Overview



Socket Creation



Socket Creation: socket

- int sockfd = socket (domain, type, protocol);
  - sockfd: socket descriptor, an integer (like a file-handle)
  - domain: integer, communication domain
    - AF_INET (IPv4), AF_INET6 (IPv6)   For this project, use AF_INET
    - AF_UNIX (communication within the local host)
    - AF_RAW (raw IP, require admin privilege)
  - type: integer, communication type   For this project, use SOCK_STREAM
    - SOCK_STREAM: reliable, connection-based
    - SOCK_DGRAM: unreliable, connectionless
  - protocol: integer   For this project, use 0
    - 0: default protocol will be used for selected domain/type
    - IPPROTO_TCP (TCP), IPPROTO_UDP (UDP)
      - Generally, IPPROTO_TCP only goes with SOCK_STREAM, and IPPROTO_UDP only goes with SOCK_DGRAM

Socket Creation: socket & perror

- Sample code

```
#include <sys/types.h>   // include many data types for system calls
#include <sys/socket.h>   // for all socket APIs
#include <netinet/in.h>   // type definitions for socket
#include <stdlib.h>   // for atoi, etc.
#include <errno.h>   // for perror and errno
…
int sockfd = socket (AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) perror ("ERROR socket: ");
```

  - Return sockfd
    - Like a file descriptor, to be used in subsequent actions
  - Error messages
    - Besides return value, system also has errno (error number)
    - You can print errno and search online to find the interpretation
    - perror does the interpretation for you

Socket Creation: bind

- int status = bind (sockfd, &addrport, size)
  - Associates address and port to the socket
    - status: 0 if successful, –1 otherwise
    - addrport: (struct sockaddr *), address for socket
    - size: the size (in bytes) of the addrport structure
  - Sample code

bzero is a C function, use memset in C++

INADDR_ANY: all local host addresses (all NICs)

```
struct sockaddr_in addr;
bzero ((char *)&addr, sizeof(addr));  // initialize addr
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons (port);  // htons convert int to big-endian
status = bind (sockfd, (struct sockaddr *) &addr, sizeof(addr));
    // type cast addr from (sockaddr_in *) to (sockaddr *)
If (status < 0) perror ("ERROR bind: ");
```

Socket Creation: bind

- int status = bind (sockfd, general-addr, size)
  - associate address to the socket
    - general-addr: (struct sockaddr *)
    - size: sizeof (addr)

```
Struct sockaddr_in addr;
bzero ((char *) &addr, sizeof(addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(port);
bind( sockfd,
      (struct sockaddr *) &addr,
      sizeof(addr) );
```

```
struct sockaddr {
  unsigned short sa_family;
  char sa_data[14];
}
```

```
struct sockaddr_in {
  short int sin_family;
  unsigned short int sin_port;
  struct in_addr sin_addr;
  unsigned char sin_zero[8];
}
```

```
// sin_port: 2 bytes
// in_addr_sin: 4-byte IP addr
// sin_zero: 8 bytes unused
```

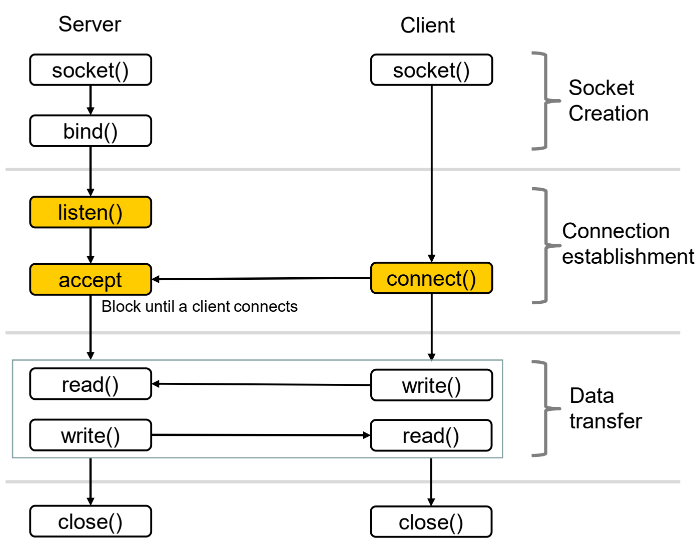
Socket Creation: bind

- int status = bind (sockfd, &addrport, size)
  - Some issues
    - Need to type cast sockaddr_in to sockaddr
      - sockaddr: general, may be used by more protocols
      - sockaddr_in: specialized for a few domains
    - htons converts "int" to unsigned short (and, in big-endian)
      - Alleviate the potential big-endian and little-endian discrepancy on different computers
  - Port number Has to be given for servere, so that clients can use the correct port
    - Choose in between 1024 to 49151
      - 1 to 1023 are reserved for special protocols, controlled by IANA
      - 49152 to 65535 are used by OS to generate random address
    - Confined pick for projects
      - [1-4] • last 4 digits of student id

Socket Creation: Close

- status = close (sockfd);
  - status: return 0 if successful, –1 if error
  - After closing a socket, the port is freed up for reuse
  - In TCP, close a connection, then re-establish it using the same port can take time, sometimes up to minutes of delay

Establish Connection



Establish Connection: Listen

- int status = listen (sockfd, backlogLimit);
  - Listen for connection requests   listen has to be and is nonblocking
  - backlogLimit: integer
    - Each connection request is queued by the system (before accept)
    - If the backlog queue is full, new connections are silently ignored
      - The client side receives a connection refused error
    - TCP/IP set a limit by SOMAXCONN, and the default value is 128
      - BacklogLimit should be <= SOMAXCONN, if it is > SOMAXCONN, it is silently truncated
      - SOMAXCONN can be changed, but needs admin privilege
    - Call listen() more than once on the same socket
      - The backlog for the same socket will be updated, other than this, it won't affect socket operation
      - But one generally do not this

Establish Connection: Client Connect

- int status = connect (sockfd, &servaddr, addrlen);
  - Client request server to establish connection   If the server specified in servaddr is not listening ⇒ returns error
  - Client creates a socket, fills in servaddr (Server's addr)
- servaddr: struct sockaddr (same as in bind)
  - By host name

```
char *h_name;   // host name
char **h_aliases;
int h_addrtype;   // e.g., AF_INET
int h_length;   // length of address
char **h_addr_list;   // list of addresses
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
hostent server = gethostbyname(…);
bzero((char *) &servaddr, sizeof(servaddr));
serv_addr.sin_family = AF_INET;
bcopy((char *) host->h_addr_list[0],
        (char *) &servaddr.sin_addr.s_addr,
        host->h_length);
servaddr.sin_port = htons(port);
```

  - By IP address

```
serv_addr.sin_addr.s_addr =
    inet_addr ("129.110.242.180")
```

```
struct sockaddr_in {
  short int sin_family;
  unsigned short int sin_port;
  struct in_addr sin_addr;
  unsigned char sin_zero[8];
}
```

```
typedef struct in_addr {
  union {
    struct {…} s_un_b;  // 4-byte
    struct {…} s_un_w;  // 2-short
    ULONG  s_addr;  // 1 word
  }              }
```

Establish Connection: Server Accept

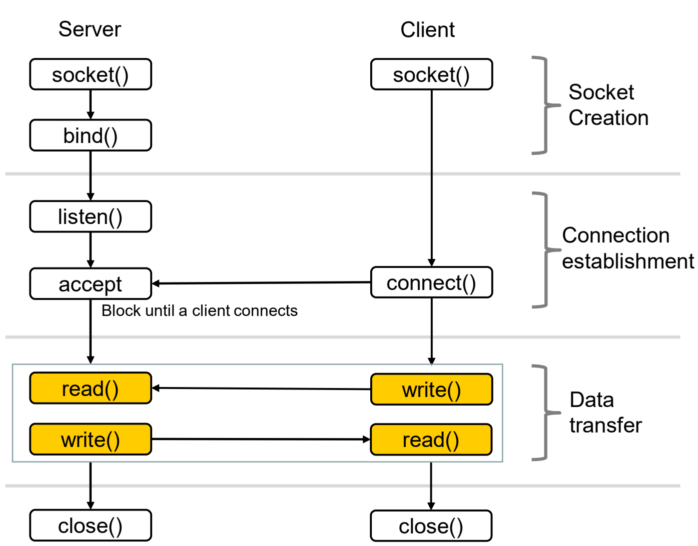
- int newsockfd = accept (sockfd, &clientaddr, &addrlen)
  - Server accept connections from multiple clients ⇒ "accept" returns a new socket descriptor for the newly established connection
  - Server will continue to listen on the original sockfd
    - listen(), once issued, will continue to listen
    - A new connection request will be put into the associated backlog, waiting for another accept call to dequeue it
    - If you have a loop to establish connections, leave listen out of the loop (though no harm with multiple listen, but it will become a problem with a large number of listen)
    - accept() is blocking
  - Struct sockaddr clientaddr
    - Returned from accept, provides client address information
    - If client IP, port are needed, can extract them from returned clientaddr
    - If only need to communicate, newsockfd is sufficient

Establish Connection: Server Accept

- int newsockfd = accept (sockfd, &clientaddr, &addrlen)
- Sample code

```
struct sockaddr_in clientaddr;
socklen_t addrlen = sizeof (clientaddr);
int newsockfd = accept (sockfd,
                        (struct sockaddr *)&cli_addr, &addrlen);
if (newsockfd < 0) perror ("ERROR accept ");
```

Socket Overview



Send and Receive Messages

- int count = send (sockfd, bufout, lenout, flags);
- int count = recv (sockfd, bufin, lenin, flags);
  - count: number of bytes sent/received (–1 if error)
  - bufout: (const void *), contains data to be sent
  - bufin: (void *), buffer for receiving data
  - lenout: length of data (in bytes) to be sent, can be less than the bufout size
  - lenin: generally the entire length of bufin
  - flags: unsigned int, configuration options, usually set to 0
    - MSG_DONTWAIT: Enables nonblocking operation
      - One flag that may be useful, it has similar behavior as setting the O_NONBLOCK flag via fcntl()
      - MSG_DONTWAIT is a per-call option, whereas fcntl() sets nonblocking for all IO on the file descriptor

IO Multiplexing

- select()
  - int select (…
    fd_set *readset, fd_set *writeset, fd_set *exceptset, …)
  - fd_set: is a bit vector
    - Represents a set of file descriptors (fd), one bit per fd
  - select()
    - Listens to read/write/exception activities on the designated IO ports and returns as soon as there is any activity happening
    - Function return is the number of ready file descriptors
      - Return = 0: could be time-out;   Return < 0, error
      - When returning, all activities happening (simultaneously) before returning will be recorded in the corresponding sets
      - The bit vector is overwritten in place, ports with the designated activities have the corresponding bits set
    - select() **blocks** till at least one of the ports is ready or time-out

IO Multiplexing

- select()

		1	1			1		1	1				. . .

readset fds = {2,3,6,8,9}
*** FD_ZERO first; otherwise
⇒ wrong bits set
number of fds = 10 (not 5)

  - fd-set
    - Set size
      - System dependent, mostly use 1024
      - Can redefine FD_SETSIZE to change the set size
        - Some systems have problems in handling > 1024 descriptors
    - ⇒ Use the standard macros for manipulating fd_sets to avoid problems caused by the uncertain size
      - int fd; fd_set set;
      - void FD_ZERO (&set); // empties the set
      - void FD_SET (fd, &set); // adds fd to the set
      - void FD_CLR (fd, &set); // removes fd from the set
      - int FD_ISSET (fd, &set) // >= 1 if fd is in the set, 0 otherwise
      - int FD_SETSIZE // the constant defines the size of the fd sets

IO Multiplexing

- select()
  - int select (int nfds, fd_set *readset, fd_set *writeset, fd_set *exceptset, struct timeval *timeout)
    - nfds : the largest file descriptor to examine
      - If you add fds = 5, 8, 13 to readset, and fds = 6, 9 to writeset, and set NULL for exceptset, then nfds should be at least 14 (0 to 13)
      - Can simply use FD_SETSIZE, which is the maximal size of the fd sets
    - readset, writeset, exceptset: the set of file descriptors to examine for readability, writability, exception status
      - Pass NULL for fdsets that are of no interest for examining
      - Note: errors are not counted as exceptions
    - timeout: specify the timeout, NULL for infinite timeout
      - struct timeval { long tv_sec; long tv_usec; };
      - Specify the seconds + microseconds for time out
    - https://www.gnu.org/software/libc/manual/html_node/Waiting-for-I_002fO.html

IO Multiplexing

- select()
  - Sample code

```
int isready (int fd1, fd2)
{ int count;
  fd_set readfds;
  struct timeval tv;
  FD_ZERO (&readfds);   // clear set readfds
  FD_SET (fd1, &readfds); FD_SET (fd2, &readfds);   // set the fd in readfds
  tv.tv_sec = 0;   tv.tv_usec = 100;   // set timeout to 100 microseconds
  count = select ((fd1>fd2?fd1:fd2)+1, &readfds, NULL, NULL, &tv);
  if (count < 0) return –1;
  if (FD_ISSET (fd1, &readfds)) printf (“Port %d is ready.\n”, fd1);
  else printf (“Port %d is not ready.\n”, fd1);
  if (FD_ISSET (fd2, &readfds)) printf (“Port %d ready.\n”, fd2);
  else printf (“Port %d is not ready.\n”, fd2);
  return 1;
}
```

Interprocess Communication: Signals

- Signal
  - Software interrupts: by software systems
    - Set a bit of in the interrupt register
  - Associated with a signal handler
  - Inform a process about the state of other processes or the OS
  - Difference from interrupts
    - Interrupts is general, including hardware and software signals
    - Hardware interrupts: IRQs (interrupt reqests)
      - Set by hardware, including clock, IO devices, CPU
      - Could assign handler in BIOS (basic IO system) or OS
      - Software interrupts are OS specific
  - Priority
    - Software signals has a higher priority than hardware signals
    - Signal may have out of sequence delivery

Signal Handling

❖ Signal system calls

- sighandler_t **signal** (int sig, sighandler_t handler);
  - typedef void (*sighandler_t) (int);   // input argument: signal number
- int **kill** (pid_t pid, int sig);

```
Another process:
Main ()
{ kill (pid, SIGRTMIN);
  sleep (1); // let previous signal handler finish
  kill (pid, SIGRTMAX);
}
```

```
void catch_ctlc (int num)
{ printf("You cannot kill this process");
  printf ("\nEnter an integer: ");
  fflush(stdout);
}

void catch_user (int num)
{ printf ("Caught signal: %d\n", num);
  printf ("\nEnter an integer: ");
  fflush(stdout);
}
```

```
main ()
{ signal(SIGRTMIN, catch_user);
  signal(SIGRTINT, catch_ctlc);
  do { printf ("Enter an integer: ");
       scanf ("%d", &input);
       printf ("Got input: %d\n", input);
     } while (input != 0);
}
```

Hardware Interrupts

Hardware interrupt example settings (map to interrupt vector)
Can be changed in BIOS or by hardware jumpers
#IRQs depends on the number of IO devices

IRQ	INT	Device
0	32	Timer
1	33	Keyboard
2	34	PIC cascading
3	35	Second serial port
4	36	First serial port
6	38	Floppy disk
8	40	System clock
10	42	Network interface
11	43	USB port, sound card
12	44	PS/2 mouse
13	45	Mathematical coprocessor
14	46	EIDE disk controller's first chain
15	47	EIDE disk controller's second chain

Software Signals in Unix

SIGHUP	1	Hang Up. Originally: a serial line drop. In modern systems: terminal closed.
SIGINT	2	A user wishes to interrupt the process (Ctl-C).
SIGQUIT	3	QUIT. Request the process to perform a core dump.
SIGILL	4	Illegal instruction (malformed, unknown, or privileged instruction).
SIGTRAP	5	Trace trap, debugger tracing event is happening (e.g., when a particular function is executed).
SIGABRT, SIGIOT	6	Abort process, usually sent by the process itself by calling abort(), can be sent from other process also.
SIGBUS	7	BUS error, such as an incorrect memory access alignment or non-existent physical address. In Linux, this signal maps to SIGUNUSED, because memory access errors of this kind are not possible.
SIGFPE	8	Floating point exception, such as division by zero.
SIGKILL	9	Forcefully terminate a process, cannot be handled by the process itself.
SIGUSR1	10	User-defined signal 1.
SIGSEGV	11	The process makes an invalid virtual memory reference, or segmentation fault.
SIGUSR2	12	User-defined signal 2.
SIGPIPE	13	Write to a pipe without a process connected to the other end.
SIGALRM	14	Notifies a process that time for the alarm() system call has expired.
SIGTERM	15	Request a process to terminate, can be caught by the process for nice termination. SIGINT is nearly identical to SIGTERM.
SIGSTKFLT	16	Stack fault. Maps to SIGUNUSED in Linux.

SIGCHLD	17	A child process terminates. One common usage is to instruct the OS to clean up the resources used by a child process after its termination.
SIGCONT	18	Continue executing after stopped, e.g., by STOP
SIGSTOP	19	Stop a process for later resumption. Cannot be intercepted by the process itself.
SIGTSTP	20	Request a process to stop temporarily (Ctl-Z). Can be captured by the process.
SIGTTIN	21	When a process attempts to read from the TTY while in the background.
SIGTTOU	22	When a process attempts to write from the TTY while in the background.
SIGURG	23	When a socket has urgent or out-of-band data available to read.
SIGXCPU	24	A process has used up its CPU in the designated duration (allow the process to save any intermediate data before it is terminated by the OS using SIGKILL).
SIGXFSZ	25	A process grows a file to a size larger than the maximum allowed size.
SIGVTALRM	26	Virtual alarm clock. May be sent by the alarm() system call. By default, this signal kills the process, but it's intended for use with process-specific signal handling.
SIGPROF	27	CPU profiling alarm clock, CPU alarm time up, may be used to implement code profiling. Similar to 27, processing will be killed, but the process should handle it.
SIGWINCH	28	When the controlling terminal (window) of a process changes size.
SIGIO, SIGPOLL	29	Input/output is now possible, used for polling. In Linux = SIGURG.
SIGPWR	30	Power failure. SIGLOST is a synonym for SIGPWR.
SIGUNUSED	31	Unused signal. In Linux, SIGSYS is a synonym for SIGUNUSED.

Blue: external requests; Violet: like blue, but cannot be captured by the process itself;
Yellow: internal problem during execution;
Brown: external problem during execution, give the process the chance to handle it;

Readings

- Section 3.1-3.5
- Section 4.1-4.2
- Project readings provided on the course webpage
  - Unix system calls
  - pthread
  - Socket programming