Concurrent Architectures - Unix: Sockets, Select & Signals

Assignment 1:

- Drop In Labs reminder
- coming soon: java -ea CardGame -S 4 | tee cardGameAnalyser
- formatting your work: why to 80 chars?

Overview (Ref: [Coulouris&al Ch 4]):

- support for concurrency in Unix
- Berkeley Sockets:
  - protocols: TCP, UDP
  - connection: socket(), bind(), listen(), accept(), connect()
  - data transfer send(), recv(); advice
- non-determinism:
  - select()
  - interrupts: motivation, H/W & S/W handling, context
  - ‘soft’ interrupts and signals
Support for Concurrency in Unix

- creating processes
  - `fork()` and `exec()` (Lab 7)

- communicating between processes
  - `pipe()` (Lab 7)

- enabling communication between computers
  - `sockets` (Lab 8)

- handling non-deterministic events
  - `select()` (Lab 8)
  - interrupts/signals

- some other stuff
  - semaphores
  - message queues (not covered in detail)
  - shared memory segments (not covered in detail)
Berkeley Sockets

- networking protocol originally provided by BSD 4.1c (1982)
- API exported by most OS's
- the principal abstraction is a socket
  - where the application attaches to the network
  - defines operations for creating connections, attaching to network, sending/receiving data and closing connection
- two forms:
  - connection oriented: Transmission Control Protocol (TCP)
  - connection-less: User Datagram Protocol (UDP)
- addressing is via an IP address + a port number [Coulouris&al Fig 4.4]

![Diagram of Berkeley Sockets]
Connection-oriented and Connection-less Sockets

(a) connection-oriented (TCP)

(b) connection-less (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_INTE, 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

- the return value is a handle (a file descriptor, fd) for the new socket
The Bind and Listen Calls

● bind 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 connection-less servers
    ◆ can use a well known port or a unique port
    ◆ bunch of routines to convert different forms of address representation (Lab 8)

● the listen call (server side)

  ■ used by connection-oriented servers to indicate that an application is willing to receive connections

  ■ `int listen(int socket, int backlog);`
  ■ `socket`: handle (fd) of newly created socket
  ■ `backlog`: number of connection requests than can be queued by the system while waiting for server to execute the `accept` call
The Accept and Connect Calls

● the accept call (server side)
  ■ after the execution of a `listen()`, the accept call carries out a passive opening (server is 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 `address` contains the client’s address

● the connect call (client side) executes an active opening of a connection
  ■ `int connect(int socket, struct sockaddr *address, int addr_len)`
  ■ call does not return until the three-way handshake (TCP) is complete (or an error)
  ■ `address`: contains remote server’s address
    ◆ client OS usually selects a random unused port
Sockets: Transferring Data & Advice

- after a connection has been made, `send()` is used to send a message on a socket:

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

- and to receive from a socket into a specified buffer:

  ```c
  int recv(int socket, char *buffer, int buff_len, int flags)
  ```

- advice:
  - read the `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 the root demon (server) processes
  - tools and info include `netstat`, `ifconfig`, `ping`, `traceroute`, `/proc`, `tcpdump` etc
Non-Determinism: Interrupts and Signals

- **Interrupt**: 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)

- The key issue is that interrupts are asynchronous to the current process

- Typically implies that some device needs service

- An interrupt is a special case of an **exception**:
  - An event that causes the processor to transition to privileged mode, with control being transferred to the kernel’s exception handler code area

- Why interrupts:
  - Allow multiple devices to be connected
    - Devices need CPU service but the CPU can’t predict when
  - External 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
  - Alternate solution: polling (Why is this bad, when is it good?)
Hardware Interrupt Handling

● 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
  ■ there is no overhead when there are no requests pending

controller signals CPU that interrupt has occurred, passes interrupt number
  ■ interrupts are assigned priorities, in order to handle simultaneous interrupts
  ■ lower priority interrupts may be disabled during service

the CPU senses (checks) interrupt request line after every instruction:
  ■ if raised, uses the interrupt number to determine which handler to start

basic (user process) state saved (as for system call)

CPU jumps to interrupt handler

when interrupt is done, process state is reloaded and the process resumes
Interrupt Context and Software Handling

- Typically there are two parts to interrupt handling
  - What must be done immediately, so that the device can keep running
  - What can be deferred until later, so that response is faster

- The first part of an handler records the context from whatever was interrupted
  - The handler is not allowed to block (or call functions that might)
  - The processor (may) have no further structure to save state
  - The handler needs to be kept as fast and simple as possible
  - Typically sets up work for second part, and flags that this in turn needs to be handled by asserting the corresponding second-level interrupt

- The deferred parts of handling are sometimes referred to as software interrupts
  - E.g. networking:
    - Time critical: copy incoming packet form the network device
    - Deferred: process packet, pass to correct application
  - E.g. timer interrupts:
    - Time-critical: increment current time-of-day
    - Deferred: recalculate process priorities
Signals

- software equivalent of (hardware) interrupts
- allows process or thread to respond to asynchronous external events
  - a process may specify its own signal handlers or may use the OS default action, typically one of:
    - ignore the signal
    - terminate all threads in the process
    - stop or resume all threads in a process
- signals provide a simple form of interprocess communication
- basic mechanism:
  - the process state includes flags for possible signals and table for actions to take
  - when a signal is posted to process, the signal pending flag is marked
  - when process is next scheduled to run, pending signals are checked and appropriate action taken
- signal delivery is not instantaneous
Signal Terminology and Issues

- posting (aka signal generation, ‘throwing’)
  - action taken when event occurs that process needs to be notified of
- delivery (aka signal handling)
  - action taken when process recognizes arrival of event
- catching
  - if a user-level signal handler is invoked, process is said to catch the signal
- pending
  - signals that have been posted but no yet delivered
- issues:
  - 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
  - only one signal handler per signal per process (can’t use in library code)
  - usually no signal queuing
Signal Usage

● write a signal handler function to handle SIGINT (Ctrl-C from keyboard)

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

● install it:

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

● block signal delivery by masking signals: sigsetmask()

● specify that signal handlers run on separate stack: sigaltstack()
  - useful if signal is caused by stack overflow

● retrieve list of pending signals: signpending()

● block process until signal is posted: sigsuspend()

● send signal to process: kill()
Process Synchronization Using Signals: sigstop.c

```c
void catch_stop(int sig) {
    printf("Handler caught signal \%d\n", sig);
    fflush(stdout); return;
}
int main(int argc, char* argv[]){
    pid_t id = fork();
    printf("Hello from Process ID \%d\n", getpid());
    if (id==0) {
        sigset_t new_mask;
        struct sigaction new_action, old_action;
        new_action.sa_handler = catch_stop;
        sigaction(SIGTERM, &new_action, &old_action);
        sigemptyset(&new_mask);
        signalsuspend(&new_mask);
        printf("process ID \%d: exiting\n", getpid());
        exit(0);
    } else {
        kill(id, SIGTERM);
        wait(NULL);
        printf("process ID \%d: child \%d has exited\n", getpid(), id);
    }
    return 0;
}
```
Programming with Signals

- **collect**: a Shell script cleans up temporary file when interrupted
  
  ```bash
  tmpdir=/tmp/collect.$$  
  trap 'cd /tmp; rm -rf $tmpdir; exit' 0 1 2 3 15
  ```

- **exception handling** in Java: done via signals
  - arguably a better method than returning error values (e.g. in C system calls)
  - any method that calls another which throws an exception must either catch it or throw it upwards
  - e.g. SingleLaneBridge.java

  - wait() or Thread.sleep() throw an InterruptedException
  - the applet’s stop() method calls Thread.interrupt() to stop the threads whenever the number of cars is reset
  - essential that the run() method in the car thread catch this properly (and then exit)
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 (!!)