Daniel **Fitzmaurice**(43961229)

1 Function Pointers

`int (*fp)(int, char*)` - Declares fp to be a pointer to a function which takes int and char* arguments and returns int
`void (*fp2[10])(double)` - Declares fp2 to be an array (of size 10) of pointers to functions taking a double parameter and returning nothing
`int (*fp3)()` - Declares fp3 to be a pointer to a function returning int. Argument types unknown and won't be checked by the compiler (Up to programmer to use this correctly).

2 Storage classes

2.1 auto

- Variable has local (automatic) extent, i.e. removed at end of block
- Permitted within a block only (i.e. not top level)
- This is the default so rarely seen

2.2 extern

- Variable/function is external to all functions, i.e. can be accessed by name by any function
- Globally accessible - linker must know about the name
- Must be defined once somewhere (can be declared anywhere)

2.3 register

- Hint to compiler to put variable in a register, otherwise like auto

2.4 static

- Name is only accessible in this file (i.e. not exported to linker)
- For variables - extent is static - variable lasts for life of program

3 5 Views of OS

1. Hardware
2. Operating System Designer
3. Application Programmer
4. End-User
5. System Administrator

4 Environment Variables

HOME - full pathname of home directory
PATH - colon separated list of pathnames to search for commands
USER - your username
SHELL - full pathname of your login shell

4.1 Local Variable

```
courseCode=CSSE2310
```

4.2 Environment Variable

```
export courseCode
```

5 Metacharacters

5.1 WildCards

***** - zero or more characters
? - any single character

5.2 Comment

- start of comment (goes till end of line)

5.3 Running Commands

& - run command in background
; - used to separate commands
'command' - substitute result of running command

5.4 Variable Substitution

$varname - substitute value of variable

5.5 SubShell

(...commands...) - execute commands in a sub-shell

5.6 Conditional sequences

|| - execute command if previous command failed
&& - execute command if previous one succeeded
A success is when the program returns an exit code 0

5.7 Redirection and Pipes

| - pipe, output of one program sent to the input of the next
> - send stdout to file
< - write stdin from file
» - append stdout to file

5.8 Quoting

Single Quotes - inhibit wildcard replacement, variable substitution, command substitution.
Double Quotes - inhibit wildcard replacement only.
Can also use backslashes.

6 Shell Scripting

6.1 Shell Arithmetic

Use `expr EXPRESSION` command. Result is printed to stdout.

6.2 Shell Conditional

Use `text EXPRESSION` or `[ EXPRESSION ]`. `EXPRESSION` follows `INT1 option INT2` where option can be: `-eq` (equals), `-ge` (greater than or equal), `-gt` (greater than), `-le` (less than or equal), `-lt` (less than), `-ne` (not equal). Result is exit code 1 for false and 0 for true.

6.3 Control Structures

For Loop

```
for name [ in word... ]
do
commands...
done
```

If words omitted, uses script arguments.

While Loop

```
while command1
do
command2
done
```

If Then Else

```
if command1
then
command2
elif command3
then
command4
else
command5
fi
```

Other Control Structures

Case. Until - do - done. Done.

7 Built-in Shell Variables

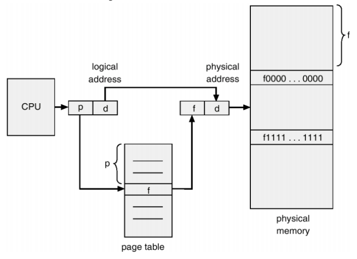
$$ - process ID of shell
$0 - name of the shell script
$1...$9 - command line arguments
$* - all the command line arguments
$# - number of command line arguments (excludes command name)
$? - exit status of last command
$! - process ID of last background command

8 Paging

One of several methods of implementing virtual memory. Divide physical memory into fixed-sized blocks called page frames (size is power of 2, usually between 512 bytes and 8192 bytes). Divide logical memory into blocks of same size called pages. Keep track of all free frames. To run a program of n pages, need to find n free frames and load program. Set up a page table to translate logical to physical addresses.

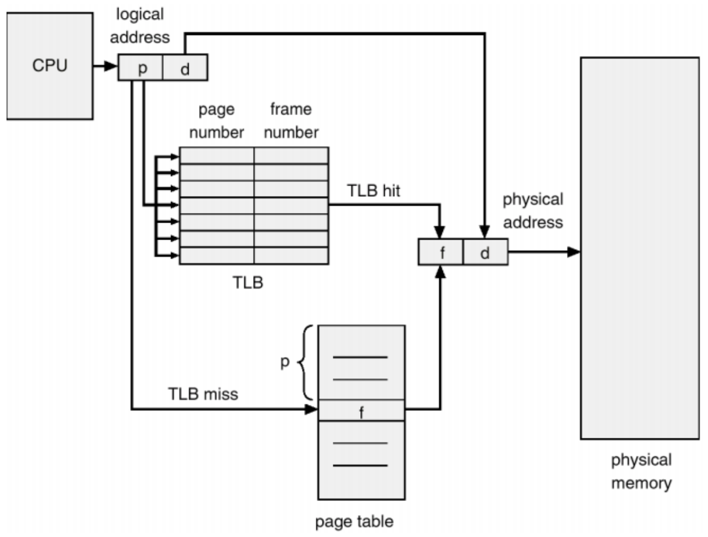
8.1 Address Translation Scheme

Page Number (p) - used as an index into a page table which contains base address of each page in physical memory
Page Offset (d) - combined with base address to define the physical memory address that is sent to the memory unit



8.2 Implementation of Page Table

Page table is kept in main memory. The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)



Page Faults

If the TLB entry doesn't exist, then load it into the table

8.3 Memory Protection

Memory protection implemented by associating protection bit with each frame.

8.4 Segmentation Faults

Segmentation Fault arises when: Accessing invalid memory page, Trying to write to a read-only page.

8.5 Shared Pages

Shared code

One copy of read-only code shared among processes. Shared code must appear in same location in the virtual address space of all processes.

Private code and data

Each process keeps a separate copy of the code and data. The pages for the private code and data can appear anywhere in the virtual address space.

8.6 User-space vs Kernel

The operating system controls the address ranges (pages) a process can use. It does not decide how that space is used. Management of those pages is the responsibility of the process (Usually vie standard libraries).

9 Memory-Related Bugs

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

10 Processes

10.1 fork

`int fork(void)` - creates a new process. Returns 0 to the child process else return child's pid.
Common for the parent to be the only one to continue forking

10.2 exit

`void exit(int status)` - exits a process (status 0 for no error)
`_exit()` - will not run the registered exit functions

10.3 atexit

`atexit(void (*func)(void))` - registers functions to be executed upon exit. Functions called in reverse order of registration.

10.4 wait

`int wait(int child_status)` - suspends process until child process terminates. Return the exit code, if child_status is `NULL` then any child process.

10.5 waitpid

`waitpid(pid, &status, options)` - can wait for a specific process. Various options (default is 0)

10.6 exec

`int execl(char *path, char *arg0, char *arg1, ..., 0)` - loads and runs executable at path with args arg0, arg1... (path is the complete path of the executable, arg0 becomes the name of the process).

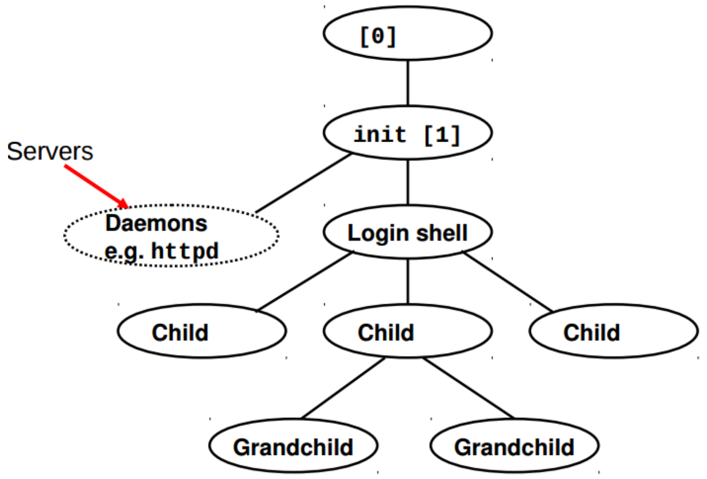
exec variations

`execl()`, `execve()`, `execv()`, `execlp()`, `execle()`, `execvp()`
l - arguments directly in call (list)
v - arguments in array (vector)
p - use PATH to find program
e - provide environment definition

10.7 Resource Sharing

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

10.8 Unix Process Hierarchy



11 Files

11.1 Unix File Types

Regular file

Binary or text file. Unix does not know the difference

Directory file

A file that contains the names and locations of other files

Links

Symbolic links to other files

Character special and block

Terminals (character special) and disks (block special)

FIFO (named pipe)

A file type used for interprocess communication

Socket

A file type used for network communication between processes

11.2 Sharing Files

A child process inherits its parent's open files. `refcnt` counts the number of references.

12 Piping

12.1 I/O Redirection

dup2

`dup2(oldfd, newfd)` - copies per-process descriptor table entry oldfd to entry newfd.

12.2 Example

```c
int fdInput[2], fdOutput[2];
if (pipe(fdInput) == 0 && pipe(fdOutput) == 0) {
    int pid = fork();
    if (pid == 0) {
        //Child
        close(fdOutput[1]);
        close(fdInput[0]);
        dup2(fdOutput[0], STDIN_FILENO);
        dup2(fdInput[1], STDOUT_FILENO);
        if (execl(processName, processName, arg1,
                arg2, NULL) == -1) {
            error(SUBPROCESS_ERROR);
        }
    } else {
        //Parent
        childPid = pid;
        close(fdInput[1]);
        close(fdOutput[0]);
        input = fdopen(fdInput[0], "r");
        output = fdopen(fdOutput[1], "w");
    }
}
return
```

13 Threads

A process may have multiple threads of control. Threads share code, data, open files etc (they have separate control flows).

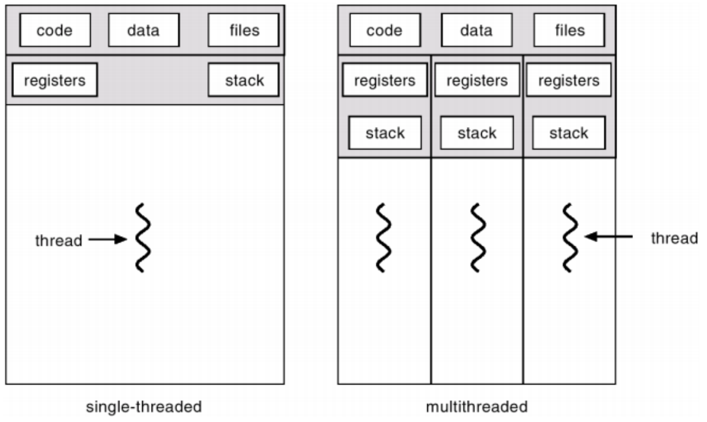
13.1 Threads vs Processes

Similarities

Each has its own logical control flow. Each can run concurrently. Each is context switched.

Differences

Threads share code and data, processes (typically) do not. Threads are somewhat less expensive than processes.



13.2 Multithreading Models

Many-to-One (User Threads)

Threads implemented in user space. OS knows nothing about them

One-to-One

Threads implemented in kernel space, one kernel thread per user thread

Many-to-Many

Hybrid model

13.3 Functions

pthread_create

`pthread_create` takes a function pointer to start the thread

pthread_join

`pthread_join` waits for a specific thread

```c
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread1, NULL);
    pthread_join(tid, NULL);
    printf("Hello from first\n");
    exit(0);
}

void* thread1(void* vargp) {
    printf("Hello from second\n");
    return NULL;
}
```

pthread_self

Returns the ID of the calling thread

pthread_exit

Exits the calling thread

pthread_cancel

`int pthread_cancel(pthread_t thread)` - sends a cancel request to thread

pthread_detach

`int pthread_detach(pthread_t thread)` - marks the thread as detached

pthread_join

`int pthread_join(pthread_t thread, void ** retval)` - waits for thread to finish

13.4 Thread Lifecycle

Ready → Running → Blocked → Terminated

14 Shared Data

Global variables - one copy per process
Local variables - one copy per thread
Static variables - one copy per process

14.1 Critical Section

A critical section of a thread is a segment of code that shouldn't be interleaved with another thread's critical section. *Note that these threads could be in different processes.* Concurrent access to shared data may result in data inconsistency.

Semaphores

Associate a semaphore S initially 1, with each shared variable (or set of shared variables). Surround corresponding critical section with `wait(S)` and `signal(S)` operations. This is a binary semaphore. Semaphore ensures mutually exclusive access to critical region.

15 Networking

15.1 Client-Server Model

Most network applications are based on the client-server model: A server process and one or more client processes.

15.2 TCP/IP

Protocol = Rules for communication
TCP = Transmission Control Protocol. Provides communication between ports on two computers (bidirectional, point-to-point, reliable, byte stream). Uses IP (Internet Protocol) to transmit small packets of data between two IP addresses.

TCP Connections

Identified by: Source IP address, Source port number, Destination IP address, Destination port number.

IP Addresses (v4)

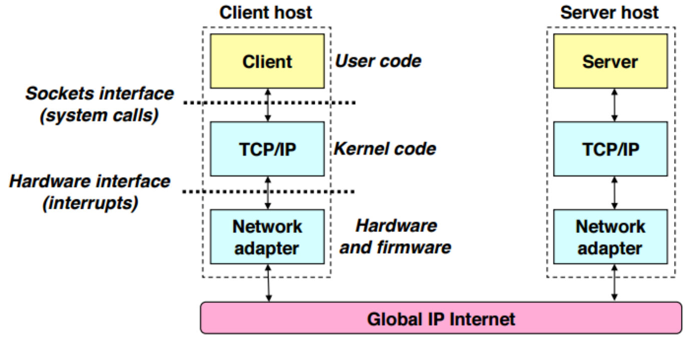
32-bit numbers. Often written in dotted decimal notation for human consumption (e.g. 130.102.2.15)

Port Numbers

16 bits: 0 - 65535. Below 1024, well known ports, reserved for standard services.

Sockets

A socket is a communication endpoint; associated with a file descriptor in UNIX - can do file I/O on socket, main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors.



`socket(...)` - create a new communication end-point
`bind(...)` - attach a local address to a socket
`listen(...)` - willing to accept connections, give queue size
`accept(...)` - wait for a connection attempt to arrive
`connect(...)` - attempt to establish a connection
`send(...)` or `write(...)` - send data over the connection
`recv(...)` or `read(...)` - receive data over the connection
`sendto(...)` - send datagram (UDP)
`recvfrom(...)` - receive datagram (UDP)
`close(...)` - release the connection
`shutdown(...)` - close down one side of connection (or both sides)

Creating a Socket

`AF_INET` - indicates that the socket is associated with Internet Protocols
`SOCK_STREAM` - selects a reliable byte stream connection (TCP)

```c
int fd = socket(AF_INET, SOCK_STREAM, 0);
if (fd < 0) {
    perror("Socket creation failed");
    exit(1);
}
```

Socket Address Structures

Generic socket address:

```c
struct sockaddr {
    sa_family_t sa_family; /* protocol family */
    char sa_data[14]; /* address data */
};
```

Internet-specific socket address:

```c
struct sockaddr_in {
    sa_family_t sin_family;
    /* address family (always AF_INET) */
    in_port_t sin_port;
    /* port num in network byte order */
    struct in_addr sin_addr;
    /* IP addr in network byte order */
    unsigned char sin_zero[8];
    /* pad to sizeof(struct sockaddr) */
};
```

Accepting a request

Blocks waiting for a connection request

```c
int listenfd; /* listening descriptor */
int connfd; /* connected descriptor */
struct sockaddr_in clientaddr;
int clientlen = sizeof(clientaddr);
connfd = accept(listenfd, (struct sockaddr*)
        &clientaddr, &clientlen);
```

Socket Options

Handy trick to allow us to rerun a server immediately after we kill it (usually have to wait 60 seconds)

```c
int optval = 1;
if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
        (const void *) &optval, sizeof(int))
        < 0) {
    perror("Unable to set socket option");
    exit(1);
}
```

Identifying the Client

The server can determine the domain name and IP address of the client

```c
char hostname[128];
/* After accept has populated clientaddr,
    clientlen */
int error = getnameinfo((struct sockaddr*)
        &clientaddr, clientlen, hostname, 128,
        NULL, 0, 0);
if (!error) {
    printf(hostname);
    printf(inet_ntoa(clientaddr.sin_addr));
}
```

Connected vs. Listening Descriptors

Listening descriptor: End point for client connection requests, created once and exists for lifetime of the server.
Connected descriptor: End point for connection between client and server, a new descriptor is created each time the server accepts a connection.

15.3 IP Addresses in C

```c
typedef uint32_t in_addr_t;
/* Internet address structure */
struct in_addr {
    in_addr_t s_addr;
    /* network byte order (big-endian) */
};
```

Handy network byte-order conversion functions:

htonl() convert uint32_t from host to network byte order
htons() convert uint16_t from host to network byte order
ntohl() convert uint32_t from network to host byte order
ntohs() convert uint16_t from network to host byte order

Function for converting between binary IP addresses and dotted decimal strings:

inet_aton(...) converts a dotted decimal string to an IP address in network byte order
inet_ntoa(...) converts an IP address in network byte order to its corresponding dotted decimal string

15.4 Testing Servers Using netcat

`nc <host> <portnumber>` - creates a connection with a server running on `<host>` and listening on port `<portnumber>`
`nc -l -p <portnumber>` - create a netcat server

15.5 Iterative Server

Iterative servers process one request as a time

15.6 Pros and Cons of ...

Process Based Design

+ Handles multiple connections concurrently
+ Simple and straightforward
- Additional overhead for process control
- Nontrivial to share data between processes (Requires IPC mechanisms)

Thread Based Design

+ Easier to share data between threads (may need mutexes/semaphores)
- Do have thread overhead

15.7 Event-Based Concurrent

Maintain a set of connected descriptors and service each as new data arrives. Repeat the following forever:

- Use the Unix `select()` function to block until:
    1. new connection request arrives on the listening descriptor, or
    2. new data arrives on an existing connected descriptor
- If 1. add the new connection to the pool of connections
- If 2. read any available data from the connection (close connection on EOF and remove it from the set)

select

`int select(int maxfdp1, fd_set *readset, NULL, NULL, NULL)` - sleeps until one or more file descriptors in the set are ready for reading.
`readset` - opaque bit vector that indicates membership in a descriptor set. If bit k is 1, descriptor k is a member of the descriptor set
`maxfdp1` - maximum descriptor in descriptor set plus 1
Returns the number of ready descriptors and sets each bit of `readset` to indicate the ready status of corresponding descriptor.

Manipulating Set Descriptors (Macros)

`void FD_ZERO(fd_set *fdset)` - turn off all bits in `fdset`
`void FD_SET(int fd, fd_set *fdset)` - turn on bit `fd` in `fdset`
`void FD_CLR(int fd, fd_set *fdset)` - turn off bit `fd` in `fdset`
`int FD_ISSET(int fd, fd_set *fdset)` - is bit `fd` in `fdset` turned on

poll

Does the same thing as select, but different interface.

16 Ways for Creating Concurrent Flows

1. Processes
    - Kernel automatically interleaves multiple logical flows
    - Each flow has its own private address space
2. Threads
    - Kernel automatically interleaves multiple logical flows
    - Each flow shares the same address space
3. I/O multiplexing with `select()`
    - User manually interleaves multiple logical flows
    - Each flow shares the same address space
    - Popular for high-performance server designs