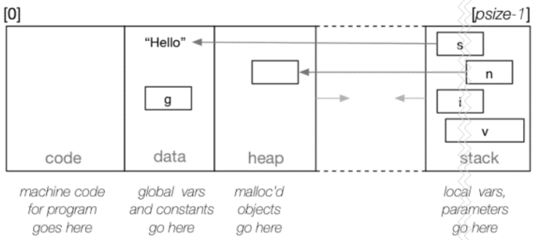
A Circus of Circuitry – A COMP1521 Notation

THEORY

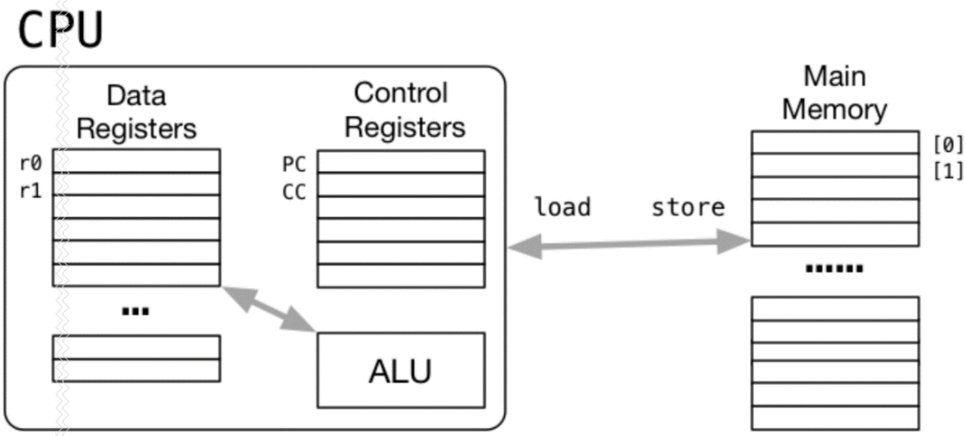
CPUs – *Level 9*

- **Processors (CPU)**
  - Control, arithmetic, logic, bit operators
  - Small set of simple operations
  - A range of ways to determine data locations
  - Small amount of very fast storage (registers)
  - Small number of control registers
  - Fast fetch-decode-execute cycle
  - Access to system bus to communicate with other components
- **Storage**
  - Memory (small and fast)
    - Very large random-addressable array of bytes
    - Can fetch single bytes, or multiple-byte chunks into CPU
    - Access time typically 0.1μs, size perhaps 64GB
  - Disk storage (large and slower)
    - Very very large block-oriented storage
    - Often on a spinning disk, 512-4KB per request
    - Access time typically 30ms, size 8TB
    - Access time nowadays 100 μs, size 512GB



- **CPU Architecture** – modern processors (CPUs) have:
  - A set of data registers
    - Fast storage used to contain data while it is being manipulated
  - A set of control registers
    - Places to control the actions of a CPU
    - Including PC – Program Counter
    - A control register which keeps track of execution
  - An arithmetic-logic unit (ALU)
  - Access to memory (RAM)
  - A set of simple instructions to:
    - Transfer data between memory and registers
    - Push values through the ALU to compute results

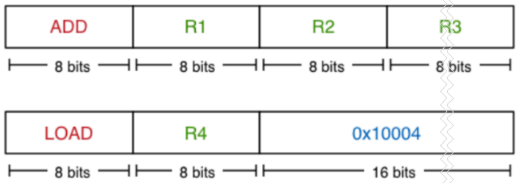
        - Make tests and transfer control of execution




```
while (1)
{
   instruction = memory[PC]
   PC++   // move to next instr
   if (instruction == HALT)
      break
   else
      execute(instruction)
}
```

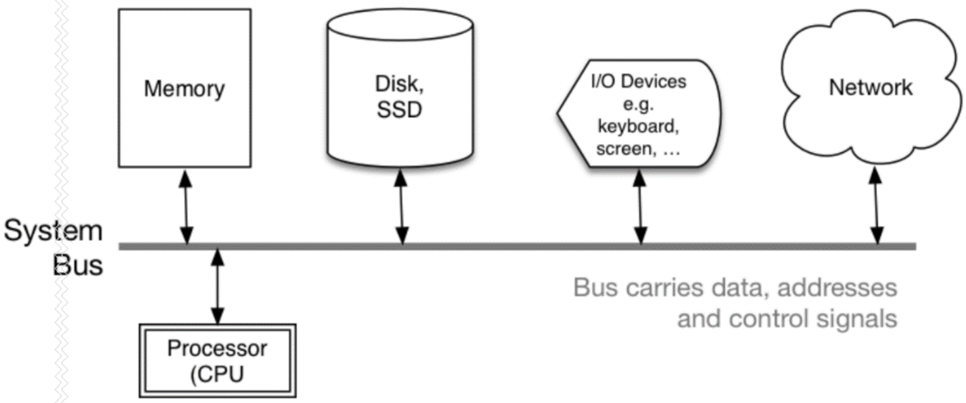
    - Execution logic looks like the right

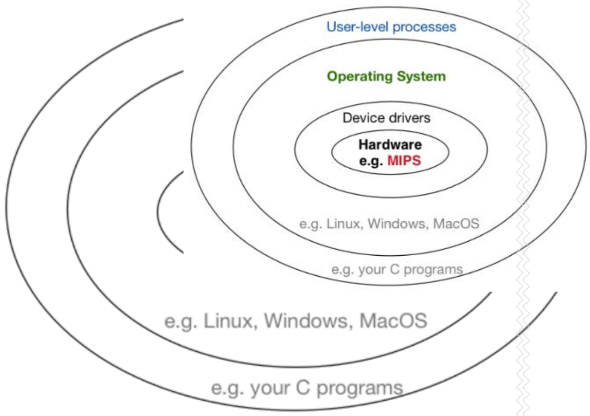
- **Instructions**
    - They are stored as strings of bits (as all things ultimately are)
        - 32-bit bit-strings
    - They contain info about:
        - What the **operator** is
        - What **registers** are involved
        - What **memory locations** are involved
    - They carry out the operation
    - They store the value (if required) in memory



Address	Content
0x100000	0x3c041001
0x100004	0x34020004
0x100008	0x0000000c
0x10000C	0x03e00008

Computer Architecture – *Like an Abacus but Amazing*

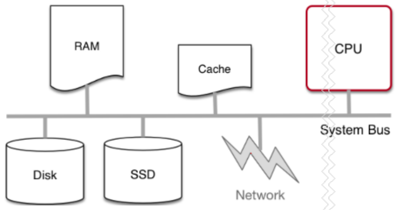




- **Computer system**
    - Hardware core surrounded by layers of software
- **Modern computer**
    - Devices connected via a system “bus”

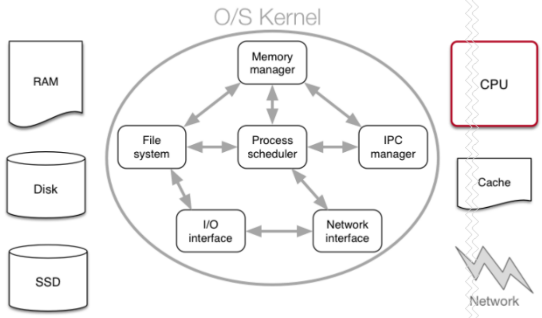
- Bus – a communication system that transfers data between the components of a computer

- **Operating systems (OSs)**
  - Provide an abstraction layer on top of hardware (provide the same view regardless of underlying hardware)
  - Have privileged access to the raw machine
  - Manage the use of machine resources (CPU, disk, memory…)
  - Provide uniform interface to access machine-level operations
  - Arrange for controlled execution of user programs
  - Provide multi-tasking and parallelism

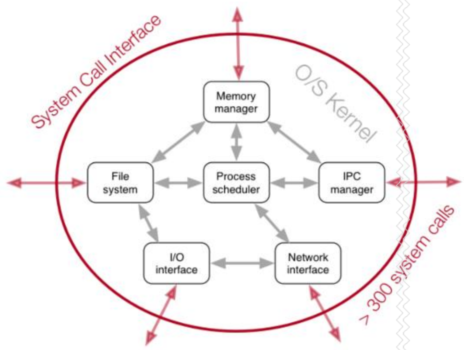


- **Operating systems 'flavours'**
  - Batch (eg Eniac, early IBM OSs)
    - Computational jobs run one-at-a-time in a queue
  - Multi-user (eg Unix/Linux, OSX, Windows)
    - Multiple jobs appear to run parallel
  - Embedded (eg Android, iOS)
  - Small, cut-down OS embedded in a device
  - Real-time (eg RTLinux)
    - Specialised OS with time guarantees on job completion

- **Modern OSs abstractions**
  - Users – who can access the system
  - Access rights – what users are allowed to do
  - File system – how data is organised
  - Input/output – transferring data to/from devices
  - Processes – active computation on the system
  - Communication – how processes interact
  - Networking – how the system talks to other systems
  - Core OS functions stem from the operating system **kernel**



- **Execution modes**
  - Privileged mode
    - Full access to all machine operations and memory regions
  - Non-privileged (user) mode
    - A limited set of operations (still Turing complete)
    - Access to only part of the memory
  - System calls allow programs to cross the privileged/user boundary in a controlled manner via well-defined requests
    - Full OSs provide 100s of system calls
      - Process management
      - File management
      - Device management
      - Information maintenance
      - Communication



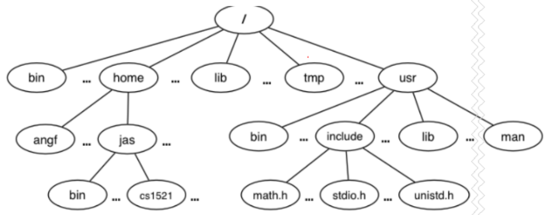
- **Libraries**
  - We resort to these when we want to do something beyond the low-level operations of system calls
  - Collections of useful functions

    - Referenced from within user programs (eg C functions)
    - Defined by #include <xxx.h>
    - Integrated with user code at 'link time'
- **Applications**
    - User-level programs which perform some useful task
    - Possibly supplied with system (ls, vim, gcc)
    - Possibly implemented by users (dcc, check, Webcms3)
    - They live in /bin, etc
    - Generally built using libraries, may also use system calls
    - Unix is unusual
        - Has a command interpreter that runs as a user-level process
        - But can invoke other user-level processes
- **System calls**
    - Invoked directly (library provided in the manual for the OS)
    - Invoked indirectly through functions in the C libraries (also documented in the manual)
    - May fail
        - This can be detected by checking return values or the global variable **errno**
        - C programs need to check and handle error themselves (provides no exception handling)
- Failed calls in C
    - fprintf(stderr, "Can't do %s", *Something*);
      exit(1)
    - perror(char **Message*);
      exit(errno); (sends precise error to shell)
    - error(*Status*, *ErrNum*, *Format*, *Expressions*, …); (Linux library function)
        - Print error message using program name, *Format* and *Expressions*
        - If *Status* is non-zero, invoke exit(*Status*) after printing message
        - If *ErrNum* is non-zero, also print standard system error message

File Systems – *The Bureaucracy*

- **File systems**
    - Provide a mechanism for managing stored data
    - Typically on a disk device
    - Allocates chunks of space on the device to files
        - A file can be viewed as a sequence of bytes
        - A directory can be viewed as a file containing references to other files
    - Allows access to files by name and with access rights
    - Arranges access to files via directories (folders)
    - Maintains information about files/directories (meta-data)
    - Deals with damage on the storage device ("bad blocks")

- Unix/Linux file system
    - “Tree structured”
    - Actually a graph after considering symbolic links
    - Used to access various types of objects
        - Files, directories, devices, processes, sockets
    - Paths can be
        - Absolute (full path from root)
        - Relative (path starts from CWD)



- File manipulation commands
    - FILE * and it’s relevant operations
    - fgets, fputs, fgetc, fputc
    - fscanf, fprintf
- Unix file system defines a range of file-system-related types
    - **off_t**
        - Offsets within files
        - Typically long and signed to allow backward references
    - **size_t**
        - Number of bytes in some object
        - (Basically an unsigned int or long long)
        - Unsigned
    - **ssize_t**
        - Sizes of read/written blocks
        - Like size_t, but signed to allow for error values
    - **struct stat**
        - File system object metadata
        - Stores info *about* a file
        - Requires ino_t, dev_t, time_t, uid_t
- **Inodes**
    - Data structures which store metadata for file system objects
        - Physical location on storage device
        - File type, file size
        - Ownership, access permissions, timestamps
    - Each file system volume (designated storage area) has a table of inodes
    - Access to a file by name requires a directory
        - Directories are lists of name/inode pairs
    - Access to files looks like this:
        - Open directory and scan for name
        - If not found, “No such file or directory”
        - If found (name, ino) access inode table inodes[ino]
        - Collect file metadata
            - Check file permissions
            - If no permission, “Permission denied”
            - Collect information, update timestamp
        - Use physical location to access device and manipulate file data
- Unix presents a uniform interface to file system objects
    - Functions and syscalls manipulate objects as a stream of bytes
    - Accessed via a file descriptor (index into a system table)

- Common operations are open(), close(), read(), write(), etc
- **int open(char *Path, int Flags)**
    - Attempt to open an object at Path according to Flags
    - Flags are constants defined in <fcntl.h>
        - O_RDONLY (open for reading)
        - O_WRONLY (open for writing)
        - Etc
        - They can be combined
- **int close(int FileDesc)**
    - Attempt to release an open file descriptor
    - If this is the last reference to object, release its resources
- **ssize_t read(int FileDesc, void *Buffer, size_t Count)**
    - Attempt to read *Count* bytes from *FileDesc* into Buffer
    - Does not check whether *Buffer* has enough space
    - Returns number of bytes read
- **ssize_t write(int FileDesc, void *Buffer, size_t Count)**
    - Attempt to write *Count* bytes from *Buffer* onto *FileDesc*
    - Does not check whether *Buffer* has *Count* bytes of data
    - Returns number of bytes written
    - Useful – can write into structs
- **off_t lseek(int FileDesc, off_t Offset, int Whence)**
    - Set the "current position" of the FileDesc (basically we move around in a file)
    - Offset is in units of bytes, can be negative
    - Whence can be:
        - SEEK_SET – set to Offset from start of file
        - SEEK_CUR – set to Offset from current position
        - SEEK_END – set file position to Offset from end of file
- **int stat(char *FileName, struct stat *StatBuf)**
    - Stores metadata associated with FileName into StatBuf
    - Includes
        - Inode number, file type and access mode, owner, group, etc
        - Size in bytes, storage block size, allocated blocks
        - Time of last access, modification, status change
    - **int fstat(int FileDesc, struct stat *StatBuf)**
        - Same but gets data via an open file descriptor
    - **int lstat(char *FileName, struct stat *StatBuf)**
        - Same but doesn't follow symbolic links

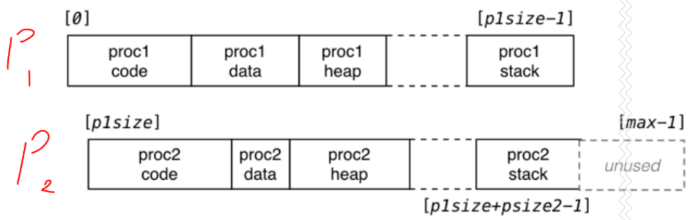
- Links
    - Hard links
        - Multiple directory entries referencing the same inode
        - The two entries must be on the same filesystem
    - Symbolic links (symlinks)
        - A file containing the path name of another file
        - Opening the symlink opens the file being referenced
    - **int mkdir(char *PathName, mode_t Mode)**
        - Create a new directory called PathName with mode Mode

            - All files in the path must exist
            - The directory we write to must be writeable to the caller
            - The directory we create must not already exist
        - The new directory contains two initial entries
            - . is a reference to itself
            - .. is a reference to its parent directory
        - Returns 0 if successful, otherwise -1
    - **int fsync(int FileDesc)**
        - Ensures that data associated with FileDesc is written to storage
        - (Usually written data is stored in memory buffers, and eventually goes to permanent storage)
        - (fsync() forces this to happen immediately

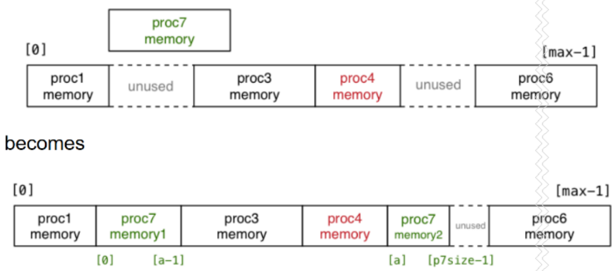
Processors and Processing – *It's a Process*

- A process is an active computation, consisting of:
    - RAM: code (read-only), data (read/write)
    - Registers: program counter (PC) and other registers
    - Other management info
- Multiple processes can be active simultaneously
    - Typically not all loaded in RAM at onces
    - Processes can be suspended (waiting)
    - Restoring a process: load code, data, registers

- If we have multiple processes…



    - We can't structure our program linearly, since the stack frames would interfere with each other
    - We thus allocate some memory for both processes
    - When we reference memory in our code, we can't do it directly, so we need to remember the base address for each process
    - We then reference specific areas of memory with $addr + \text{p1size}$ instead of $addr$
        - This can be automatically mapped during execution
    - Remember base address for each process (process table)
        - When process starts, load base into mapping hardware
        - Interpret every $addr$ as $base + addr$
- If we add a new process…

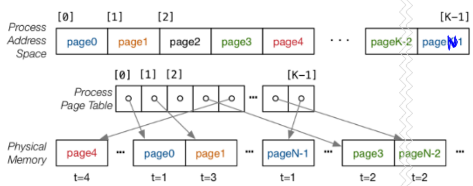


    - Slot it in a free chunk of memory that's big enough
    - Otherwise, split our process into two smaller chunks of memory

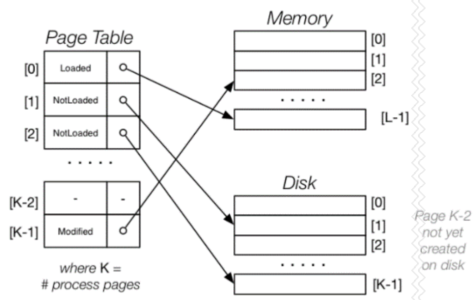
- **Memory Management**

- It makes sense to standardise chunks of memory
  - Call each chunk of address space a **page**
  - All pages are the same **size** P
  - Page i has addresses A in the range $i * P \leq A < (i+1) * P$
- **Virtual addresses** and address mapping tables
  - Each process has an array of page entries
  - Each page entry contains the start address of one chunk
  - Can compute the index of relevant page entry by A/P
  - Can compute the offset within the page by $A\%P$
- Eg: Pages p0@5000, p1@3000, p2@1000, page size 1000
  - Virtual address 0 refers to physical address 5000
  - Virtual address 128 refers to physical address 5128
  - Virtual address 1500 refers to physical address 3500
  - Virtual address 2200 refers to physical address 1200
- If pages are of size $Pagesize == 2^n$...
  - Computing page number and offset becomes very efficient
  - We simply use bitwise operators with two bitmasks (for pageno and offset)

- **Virtual Memory**
  - A side effect of this mapping is that we don't need to load all of our process's pages upfront (like we would in MIPS)
    - We start with a small memory footprint (ie just main and stack top)
    - We load new process address pages into memory as needed
    - Grow up to the size of the physical memory
  - We can:
    - Divide process memory space into fixed-size pages
    - Load process pages into physical memory
  - This is virtual memory
    - We map virtual to physical memory in a table

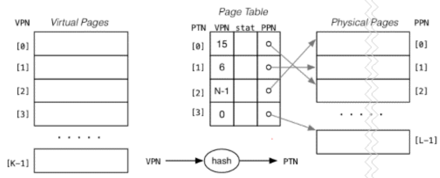


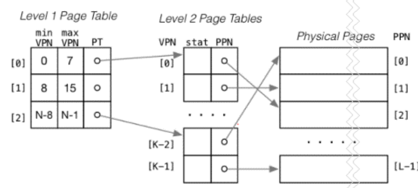
- Implications
  - In any given time, a process is likely to access only a fraction of its pages
  - This is known as the **working set model**
  - We only need to hold the working set at a given time
  - If each process has a relatively small working set, it can hold pages for many processes simultaneously
  - *The process address space can be larger than immediate physical memory*
- Where do we "load" pages from?
  - Code, global data from executable file stored on disk
  - Dynamic (heap, stack) data is created in memory
  - Note: transferring between disk→memory is very expensive



- **Pre-process page table**
  - Can be considered a struct of structs, stored in memory
  - Allows us to keep track of what is and is not loaded
  - Requesting a non-loaded page generates a **page fault**
    - We often keep a **free list** to keep track of free frames
    - If there are no free frames, we either *suspend* the request or *replace* a currently loaded page

- Page replacement
    - When a page is replaced
        - If it's been modified since loading, save to disk (requires flags)
        - Grab its frame number and give it to the requestor
    - Which frame should be replaced
        - Define a "usefulness" measure for each frame
        - Grab the frame with the lowest usefulness. When replacing…
            - Prefer pages that are read-only (no need to write to disk)
            - Prefer pages that are unmodified (no need to write to disk)
            - Prefer pages that are only used by one process
            - "A page not used recently may not be needed again soon" (LRU replacement)
    - **Thrashing**: constantly swapping pages in and out of memory
        - The working set model and LRU helps avoid this
- Page replacement strategies
    - **LRU**
        - Least recently used
    - **FIFO**
        - Page frames are entered into a queue and page replacement uses the frame from the front of the queue
    - **Clock sweep**
        - Uses a reference bit for each frame
        - Maintains a circular list of allocated frames
        - Uses a "clock hand" which iterates over page frame list
            - All referenced pages are skipped and reset
        - First-found unreferenced frame used
- Page table strategies (we only need in practice some $n \ll K$ page table entries at one time
    - Hashing
        - Make a page table
        - Hash it (map several pages to one spot to save space)
    - Multi-level page tables
        - One table of tables
        - Ie entries 0 to 7 in one table, 8 to 15 in another, etc
        - It's unlikely we use all at once





- **Cache memory**
    - Small, fast memory close to CPU
    - Holds parts of RAM we think will be heavily used
    - Transfers data to/from RAM in blocks
    - Memory reference hardware checks cache first
        - If not there, gets from RAM
    - Similar replacement strategies similar to virtual memory
- Memory management hardware
    - Address translation is done by MMU
    - TLB – translation lookaside buffer
- **Process**
    - An instance of an executing program

- Each process has an execution state defined by
  - Current execution point
  - Current values of CPU registers
  - Current contents of virtual address space
  - Information about open files, sockets, etc
- To manage processes, maintain
  - Process page table
  - Process metadata
- OS ideology
  - Multiple processes are active "simultaneously"
    - The processes have control-flow independence and private address space

- **Control-flow independence**
  - Each process executes as if the only process running on the machine
  - If there are in fact multiple processes running on the machine
    - Each process uses the CPU until pre-empted or exits
    - Then another process uses the CPU until it is pre-empted
    - Eventually the first process will get another run
    - (Effectively a priorityQ)
  - What causes pre-emption?
    - If it runs "long enough" and the OS replaces it
    - It attempts to perform a long-duration task
  - On pre-emption
    - The process's entire **context** is saved
      - Static information (code, data)
      - Dynamic state (heap, stack, registers)
      - OS-supplied state (environment variables, stdin, stdout)
    - The process is flagged and suspended
    - It is places on a process (priority) queue
    - On resuming it is restored
      - Contexts are entirely switched (**context switch**)
  - Non-static process context is held in a **process control block (PCB)**
    - Identifier (unique process ID)
    - Status (running, ready, suspended, exited)
    - State (registers)
    - Privileges (owner, group)
    - Memory management info (page table reference)
    - Accounting (CPU time used, amount done)
    - I/O (open file descriptors)
  - The OS scheduler maintains a queue
  - The OS maintains a table of PCBs
- Unix/Linux Process
  - Commands
    - sh - For creating processes via object-file name
    - ps - Show process info
    - w - Show per-user process info
    - top - Show high-CPU-usage process info

            - kill - Send a signal to a process
    - PCB info
        - pid – process id
        - ruid, euid – real and effective user id
        - rgid, egid – real and effective group id
        - Cwd
        - Accumulated execution time
        - User file descriptor table
        - Info on how to react to signals
        - Pointer to process page table
        - Process state
    - **Process ID (PID)**
        - A positive integer, unique
        - Type pid_t
        - Process 0 is the idle process (always runnable)
            - A 'kernel artefact'
            - Not a real process
            - Needed to ensure there is always at least one process
        - Process 1 is **init** ("the system")
        - Low-numbered processes are system-related
        - Regular processes are in (300, maxPID (maybe 2^{16}))
    - 'Families'
        - Each process had a parent process (usually that which created it)
        - A process may have child processes
        - Process 1 created at system startup
            - *If a process' parent dies, it is inherited by process 1*
    - **Process groups**
        - Each group is associated with a unique **PGID**
        - Type pid_t
        - A child belongs to the process group of its parent
        - A process can create its own process group or move to another one
        - Allow the OS to keep track of groups working together
            - Signals sent to related processes
            - Management of processes for job control
            - Management of processes within pipelines
    - **System calls** (and failure)
        - Use wrapper functions
        - Same arguments/returns as system call
        - Catches and reports the error
        - Only ever returns with a valid result
        - Not always appropriate (open() is best handled by caller)
- Unix/Linux process-related system calls
    - **pid_t fork(void)**
        - Requires #include <unistd.h>
        - Creates new process ***by duplicating the calling process***
        - New process is the child
        - Child has different process ID

- Returns
  - In the child, fork() returns 0
  - In the parent, fork() returns the pid of the child
  - If the call fails, fork() returns -1
- Child inherits copies of parent's address space and open file descriptors
- Wrapper function for fork():

```
pid_t Fork() {
    pid_t pid;
    if ((pid = fork())<0) {
        perror("fork() failed");
        exit(1);
    }
    return pid;
}
```

- **void _exit(int status)**
  - Terminates current process
  - Closes any file descriptors
  - A SIGCHLD signal is sent to parent
  - Returns **status** to parent (via wait())
- **void exit(int status)**
  - Terminates current process
  - Triggers any functions registered as atexit()
  - Flushes stdio buffers, closes open FILE *s
  - Then behaves like _exit()
- **void atexit(void (*func)(void))**
  - Executes the function at exit
- **pid_t getpid() / getppid()**
  - Requires #include <sys/types.h>
  - Returns the process ID / parent process ID of the current process
  - Also getpgid(), setpgid()
- **pid_t waitpid(pid_t pid, int *status, int options)**
  - Pause current process until process pid changes state (finishing, stopping, restarting)
  - Ensures that child resources are released on exit
- **pid_t wait(int *status)**
  - Pauses until one of the child processes terminates
- **int kill(pid_t ProcID, int SigID)**
  - Requires #include <signal.h>
  - Send signal SigID to process ProcID
  - Various signals, eg:
    - SIGHUP – hangup detected on controlling terminal/process
    - SIGINT – interrupt from keyboard (ctrl+C)
    - SIGKILL – kill signal
    - SIGILL – illegal instruction
    - SIGFPE – floating point exception
    - SIGSEGV – invalid memory reference

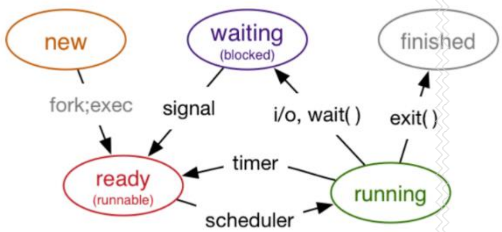
        - SIGPIPE – broken pipe
  - **int execve(char *Path, char *Argv[], char *Envp[])**
    - Transforms current process by executing Path object
      - Path must be an executable, binary or script (starting with #!)
    - Passes array of strings to new process
      - Both arrays terminated by a NLL pointer element
      - Envp[] contains strings of the form key=value
    - Most of the original process is lost
      - New virtual address space, signal handlers reset, etc
    - New process inherits open file descriptors from original process
- **Zombie process**
  - A process which has exited but signal not handled
  - All processes become zombie until a SIGCHLD handled
  - Parent may be delayed, but usually resolved quickly
  - Long-term zombies created when a bug in parent causes it to ignore SIGCHLD
  - Zombies occupy a slot in the process table
- **Orphan process**
  - When parent exits, orphan is given pid=1 as its parent
  - pid=1 always handles SIGCHLD when process exits

- **Process Control Flow**
  - Flow
    - Fetch instruction from memory[PC], PC++
    - Decode and execute instruction
      - If jump-type instruction, PC = new address
      - If regular instruction, carry out
    - Repeat above
  - System calls
    - Typically operate via exceptions
    - Generates an exception
      - Control transferred to exception handler
      - Carries out system-level operations
      - Returns control to the process
    - Effect is like a normal function call
  - Signals
    - Can be generated from a variety of sources
      - From another process via kill()
      - From the OS
      - From within the process
      - From a fault in the process
      - From a device
    - Processes can define how they want to handle signals

- **Signal handlers**
  - Functions invoked in response to a signal

- Knows which signal it was invoked by
- Ensures the invoking signal is blocked and carried out any appropriate action
- **SigHnd signal(int SigID, SigHnd Handler)**
  - *Defines **with what function** a process will handle a signal*
  - Requires <signal.h>
  - SigID is an OS-defined signal
  - Handler is one of:
    - SIG_IGN (ignore)
    - SIG_DFL (default handler)
    - A user defined function to handle SigID signals
- **int sigaction(int sigID, struct sigaction *newAct, struct sigaction *oldAct)**
  - sigID is an OS-defined signal
  - newAct defines how signal should be handled
  - oldAct saves a copy of how signal was handled
  - if newAct.sa_handler ==
    - SIG_IGN, ignore
    - SIG_DFL, default
  - struct sigaction has:
    - A pointer to a handler function
    - A mask, specifying signal to block
      Flags to modify how signal is treated

- **Interrupts**
  - *Signals which cause normal process execution to be suspended*
  - An **interrupt handler** then carries out tasks related to interrupt
    - Control is then returned to original process
  - Egs
    - When data is fetched from the disk, interrupt and place data in a buffer
    - If a process is pre-empted
- **Exceptions**
  - *Like interrupts but for internal factors (computation)*
  - Egs
    - Division by 0
    - Failed malloc call
- **Muti-tasking summary**
  - Multiple processes “active”
  - Processes not *executing* simultaneously (although could with multiple CPUs)
  - Processes are a mixture of
    - Blocked waiting on signal
    - Runnable
    - One is running (each CPU)
  - Appearance of multiple simultaneous processes
    - Swap process after one runs for a defined time
    - Processes are **pre-empted**
    - System scheduler selects new process

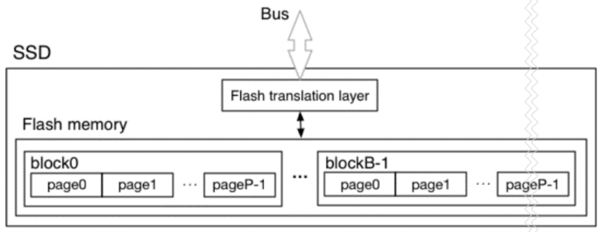


- **Scheduling**
  - Processes organised into priorityQs
  - Determined by several factors:

        - System processes > user processes
        - Long running <
        - Memory intensive <
        - Some processes suggest their own priority

Device Management – *The Heavy Hardware*

- I/O devices
    - Allow programs to communicate with “the outside” world
- Memory-based data
    - Fast random access via virtual address
    - Transfer data in bytes, words, etc
- Device data
    - Must slower access
    - Random or sequential
    - Often in blocks (128B, 512B, 4KB etc)
- Hard disk
    - Address by track and sector
    - Around 0.1ms transfer
- Solid State Disk (SSD) characteristics
    - High capacity (GB), high cost, reads faster than writes
    - Pages 512B..4KB, blocks 32..128 pages, R/W page-at-a-time
    - Pages updates by erase then write
    - Average read/write time in order of 10^{-5}



- **Device Drivers**
    - Each type of device has a unique access protocol
        - Special control and data registers
        - Locations for data to be R/W
    - Device drivers – code chunks to control an i/o device
        - Core OS components
        - Sometimes written in assembler
    - Typical protocol to manipulate
        - Send request
        - Receive interrupt when completed
- Memory mapped I/O
    - OS defines special memory address
    - User programs perform i/o by getting/putting data into memory
    - Virtual addresses associated with
        - Data buffers
        - Control registers
    - Advantages:
        - Can use existing memory to access logic circuits
        - Can use full range of CPU operations
- Devices on Linux/Unix

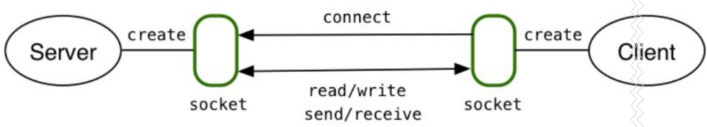
- Device "files" can be accessed via file system under "/dev"
  - dev/diskN hard drive
  - dev/ttyN terminal device
  - dev/ptyN pseudo-terminal device

    and also lol…
  - dev/mem the physical memory of the computer
  - dev/null data sink / empty source
  - dev/random stream of pseudo-random numbers

- **int ioctl(int FileDesc, int Request, void *Arg)**
  - Manipulates parameters of special files
  - Request is a device specific request code
  - Arg is an integer modifier or pointer to data block

Networking – *Gotta Make Those Connections*

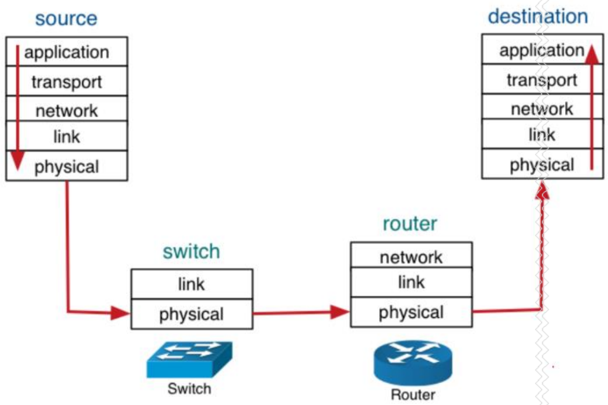
- **Sockets**
  - Unix/Linux provides sockets to communicate processes that may be on difference systems
    - Consider them to be "a door"
    - Both endpoints need to create one



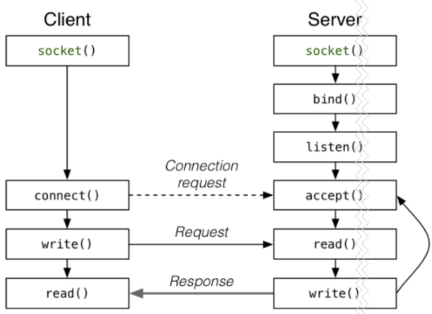
  - Socket – an end-point of a channel
    - Either locally (Unix domain) or network-wide (internet domain)
  - Server creates a socket, then
    - Binds to an address
    - Listens for connections from clients
  - Client creates a socket then
    - Connects to server using an address
    - R/W to server via socket
  - **int socket(int Domain, int Type, int Protocol)**
    - Requires #include <sys/socket.h>
    - Creates a socket using:
      - Domain – communications domain
        - AF_LOCAL (local host)
        - AF_INET (over the network)
      - Type – semantics of communication
        - SOCK_STREAM (sequenced, reliable communications stream)
        - SOCK_DGRAM (connectionless, unreliable packet transfer)
      - Protocol – communication protocol
        - Many exist
  - **int bind(int Sockfd, SockAddr *Addr, socklen_t AddrLen)**
    - Associates open socket with an address
    - For Unix, address is a pathname
    - For Internet, address is IP and port number
  - **int listen(int Sockfd, int Backlog)**

        - Wait for connections on socket Sockfd
        - Allow at most Backlog connections to queue up
    - **SockAddr = struct sockaddr_in**
        - sin_family – domain (AF_UNIX or AF_INET)
        - sin_port – port number
        - sin_addr – structure containing host address
        - sin_zero[8] – padding
    - **accept(int Sockfd, SockAddr *Addr, socklen_t *AddrLen)**
        - Sockfd has been created, bound and is listening
        - Blocks until a connection request is received
        - Sets up a connection between client/server after connect()
    - **connect(int Sockfd, SockAddr *Addr, socklen_t *AddrLen)**
        - Connects the socket Sockfd to address Addr
        - Assumes that Addr contains a process listening appropriately
- **Networks**
    - Interconnected collections of computers
    - Flavours
        - **Local area networks** (within a physical location) (**LAN**)
        - **Wide area networks** (geographically dispersed) (**WAN**)
        - **Internet** (global set of WANs)
    - Basic requirements
        - Get data from machine A to machine B
        - A and B may be separated by 100s of networks and devices
    - **How to achieve**
        - Need a unique address for destination
        - Identify a route
        - Process at intermediate nodes
        - Follow certain protocols
    - **How a file is sent**
        - File data divided into packets by source device
            - Packets are small fixed-size chunks of data with headers
        - Passed across physical link
            - Wire, radio, optic fibre
            - May not actually be “physical” physical lol
        - Passed through multiple nodes
            - Each node redirects to another node
        - Packets reach destination
        - Re-ordering, error-checking, buffering
        - File received by receiving process or user

- **The Internet**
    - Concept
        - Millions of connected devices
        - Communication links
            - Fibre, copper, radio satellite
        - Packet switches
            - Routers, network switches



- **The 5-layer Model for the Internet**
    - **Lowest (least abstract)**
    - Physical layer
        - Bits
    - Link layer
        - Ethernet, MAC addressing, CSMA
    - Network layer
        - Routing protocols (IP)
    - Transport layer
        - TCP – Transmission Control Protocol
            - Wait for a signal (connection established and held)
        - UDP – User Datagram Protocol
            - Keep sending, minimal checks in place, no guarantees
    - Application layer
        - NWS, HTTP, email, FTP
    - **Highest (most abstract)**
- **Protocols** – effectively 'rules and regulations'
    - Govern all communication activity on the network
        - Format and order of messages sent/received
        - Actions taken on transmission/receipt
    - Defined in all the layers
        - Link: PPP (point-to-point)
        - Network: IP (internet protocol)
        - Transport: TCP (transmission control), UDP (user datagram)
        - Application: HTTP, FTP SSH, POP, SMTP
- **Client-Server Architecture**
    - Server is a data provider
        - Process that waits for requests
        - Always-on host, permanent IP
        - Possibly using data centres / multiple CPUs
    - Client is a data consumer
        - Sends requests to server, collects response
        - May be intermittently connected
        - Does not communicate with other clients

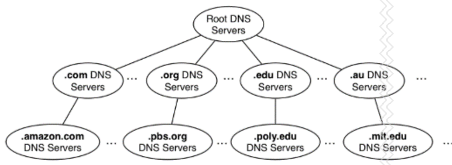


- **IP Addresses**
    - Unique identifiers for host on a network
    - Given as a 32-bit identifier (dotted quad)
        - 127.0.0.1 (refers to local host)
    - IP addresses are assigned by the sys admin entering into local registry
- **The HTTP Protocol**

- HTTP = HyperText Transfer Protocol
    - Drives the web
    - Message types:
        - URLs (requests)
        - Web pages (responses)
    - Message syntax:
        - Headers and data
    - Client-server model:
        - Client = web browser
        - Server = web server

- **URL components**
    - Protocol
    - Host
    - Port
    - Path
    - Query (everything after the '?')

- Server Addresses (DNS)
    - **Domain Name System** – *provides name to IP mapping*
        - Ie www.cse.unsw.edu.au → IP



    - Effectively a database of mappings
        - Name servers cooperate to achieve this
    - Name resolution (two styles)
        - **Iterated query**
            - Client contacts name server X
            - Response: "I don't know, but ask name server Y" OR "Here it is"
        - **Recursive query**
            - Client contacts name server X
            - X contacts Y, Y contacts Z… until resolve
    - Management of servers
        - Top-level domain (TLD) name servers
        - Authoritative name servers (within an organisation)
        - Local (default)
    - The Unix/Linux **host** command does this too

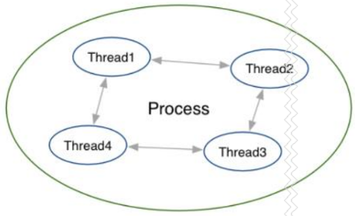
- Transport Layer deals with
    - Data integrity
        - Some apps (file transfer) require 100% reliable transfer
        - Other apps can tolerate some loss
    - Timing
        - Some appes (eg networked games) require low transmission delay
    - Throughput
        - Some apps require minimum throughput

        - Others ("elastic apps") can use whatever
    - Security
        - Some require encrypted transmission

Multiple Processes – *Threading the Needle*

- **Parallelism**
    - Multiple computations executed simultaneously
    - Eg multiple CPUs, one process on each CPU
    - Eg data vector, one processor computes each element
    - Eg SIMD (single instruction, multiple data) like above
- **Concurrency**
    - Multiple processes running (pseudo-) simultaneously
    - Eg single CPU alternating between processes
        - Can be achieved from processes
        - This requires kernel intervention
        - Each process has a significant amount of state
        - Requires system-level mechanisms
    - Increases system throughput
        - If one process is delayed, others can still run
        - Use all CPUs if multiple
    - **Effects of poorly controlled concurrency**
        - **Nondeterminism** – same code, different runs, different results
        - **Deadlock** – a group of processes wait for each other
        - **Starvation** – one process keeps missing access to a resource
        - Other: eg bank withdrawal conundrum
    - Concurrency control
        - Provide correct sequencing of interactions between processes
        - Coordinate semantically-valid access to shared resources
        - Shared memory based scheme
            - Uses shared variable, manipulated atomically
            - Blocks if access unavailable, decrements once available
        - **Message passing** based scheme
            - Processes communicate by sending/receiving messages
            - Receiver can block waiting for message to arrive
            - Sender may block waiting for message to be received
- **Producer-consumer problem**
    - Producer produces (puts item into an array)
    - Consumer consumes (takes item from an array)
    - We need a mechanism for a process to pause itself, and a mechanism for signalling for a process to wake up
- **Semaphores**
    - Shared memory objects to prevent these issues (literally just a shared variable)
        - Can be thought of as a count of available resources
    - **Init(Sem, InitValue), while(Sem), signal(Sem)**
        - Init – set value

    - Wait – if sem>0, decrement, else wait until sem>0
    - Signal – increment sem
    - Note: a series of Linux commands for this exist
- Threads – alternate method for concurrency
  - Threads exist *within* a parent process (processes independent)
  - Threads share parent process state (processes own)
  - All threads share an address space (processes own)
  - Threats communicate via shared memory (processes use IPC)
  - Content switching between threads is cheaper than with processes

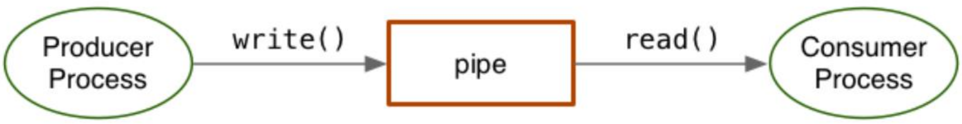


- Linux/Unix thread commands (#include <pthread.h>)
  - **Int pthread_create(pthread_t *Thread, pthread_attr_t *Attr, void *(*Func)(void *), void *Arg)**
    - Creates a new thread with specified attributes
    - Thread info in *Thread
    - Starts by executing Func() with Arg
  - **pthread_t pthread_self(void)**
    - Returns pthread_t for current thread
    - Analogous getpid()
  - **int pthread_equal(pthread_t t1, pthread_t t2)**
    - Compares two threads
    - Returns non-zero if same thread
  - **int pthread_join(pthread_t T, void **value_ptr)**
    - Suspend execution until threat T terminates
    - Exit value placed in value_ptr
    - If T has exited already, does not wait
  - **void pthread_exit(void *value_ptr)**
    - Terminate a thread and clean up
    - Store return value in value_ptr
- MapReduce
  - A model for manipulating very large data on a large network of nodes
  - Map – filter data and distribute to nodes as (key, value) pairs
    - Each node receives a set of pairs with common keys
  - Reduce – nodes perform calculation on received data items
    - Outputs from the nodes combined

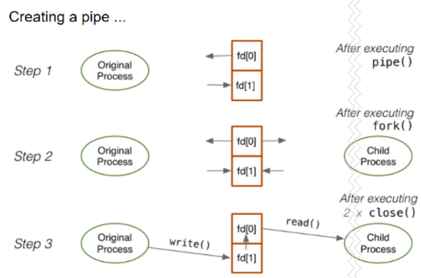
- Interacting processes
  - Processes can interact via:
    - Signals, signal handlers
    - Accessing the same resource
    - Pipes (stdout of process A goes into stdin of process B)
    - Message queues
    - Sockets
  - Issues
    - Eg two processes writing to the same file 'simultaneously'
- **File Locking**
  - **int flock(int FileDesc, int Operation)**
  - Operations:
    - LOCK_SH (acquire share lock)

        - LOCK_EX (acquire exclusive lock)
            - Blocks if a process tries to acquire shared lock
            - Will fail if already locked
        - LOCK_UN (unlock)
        - LOCK_NB (operation fails rather than blocking)
    - Does not return until lock available
    - Only works if all processes accessing file use locks

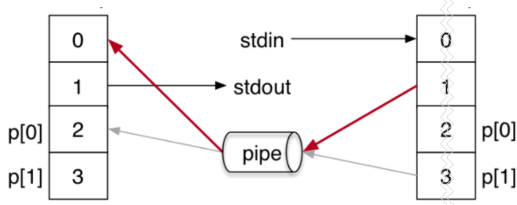
- **Pipes**
    - A style of process interaction
    - Producer process writes to a byte stream, consumer process reads from same byte stream
    - Provides buffered i/o between producer and consumer
        - Producer blocks when buffer full
        - Consumer blocks when buffer empty
    - Pipes are bidirectional (unless processes close one file descriptor)



    - **int pipe(int fd[2])**
        - Open two file descriptors (shares by proceses)
        - fd[0] is opened for reading, fd[1] for writing
    - Creating a pipe would be followed by fork() to create a child process
        - Both have copies of fd[]
        - One can write to fd[1], the other can read from fd[0]

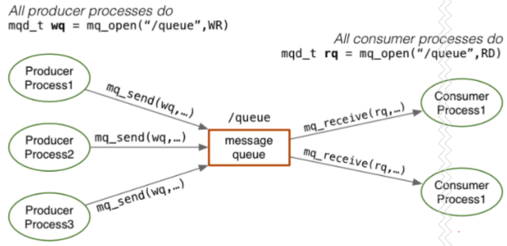


    - **FILE *popen(char *Cmd, char *Mode)**
        - Analogous to fopen()
        - Cmd is passed to shell for interpretation
        - Returns FILE *, which can be read/written based on Mode (or NULL if can't establish)
    - Only works because both processes share an address space (both can access p[])
      What if we execve() in the child? How can we still access the pipe?
        - Connect stdout and p[1] in the parent process
        - Connect stdin and p[0] in the child
        - *Then* exec (**parent can send data to child**)
        - *Swap 'em for child sending data to parent*



    - Connecting via dup2()
        - **dup2(fd1,fd2)**
        - Copies file descriptor fd1 onto fd2
        - fd's are just indexes into a table of structures
        - **dup(fd1)** copies fd1 onto the "next free" fd

- **Message queues**
  - Provides a mechanism for unrelated processes to pass information along a buffered channel shared by many processes
  - Processes connect to message queues by name
  - Requires #include <mqueue.h>
  - **mqd_t mq_open(char *Name, int Flags)**
    - Creates a new message queue, or opens existing one
    - Flags same as fopen() (eg O_RDONLY)
  - **int mq_close(mqd_t *MQ)**
    - Finish accessing message queue MQ
    - It continues to exist
  - **int mq_send(mqd_t MQ, char *Msg, int Size, uint Prio)**
    - Adds message Msg to the queue MQ
    - Prio gives priority
    - If MQ is full, blocks until MQ space available
  - **mq_timedsend()**
    - Waits for specified time if MQ full
    - Fails if still no space on MQ after timeout



CRAFT

Bits and Bytes – *Delicious!*

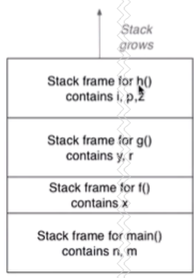
- One byte is 8 bits
  - int = 4 bytes = 32 bits
- C bitwise operators
  - **& (AND)**
    - Goes through every digit and applies the operator to each matching digit
  - **| (OR)**
    - Likewise
  - **^ (XOR)**
    - Result is 1 if the compared numbers are different, else 0
  - **~ (NEG)**
    - Negation – converts all zeroes to ones and ones to zeroes
  - **<< (left shift)**
    - Sends digits to the left by a given amount and replaces with 0s
      - Eg: "01011011 << 3" gives 11011000
  - **>> (right shift)**
    - Sends digits to the right by a given amount and replaces with 0s**
    - **unless the quantity is signed, then it is replaces with the first digit
      - Eg: "01011011 >> 3" gives 00010110

- Representing numbers
    - **Hexadecimal**
        - Each number is 4 bits
        - Eg 4 = 0100
        - Eg F = 1111
        - When writing hex in C, begin it with 0x
            - Eg 0x71 for 113
    - **Octal**
        - Each number is 3 bits
        - Eg 5 = 101
        - Eg 6 = 110
    - **Binary**
        - Each digit is one bit
        - Eg 1 = 1
        - Eg 0 = 0
        - When writing binary in C, begin it with 0b
            - Eg 0b1101 for 13

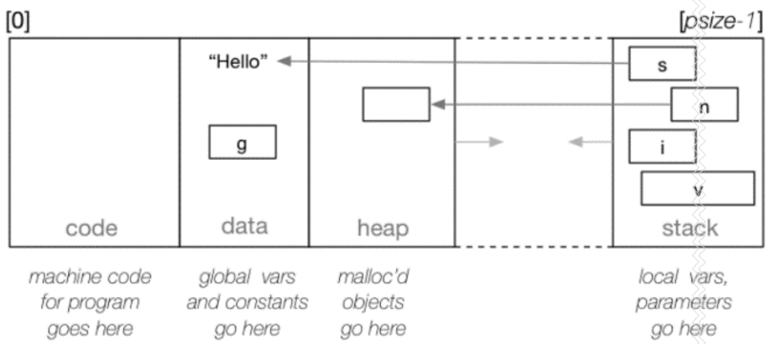
Data Representation – *Let Me Lay it All Out for Ya*

- Typical memory usage for a given C code
    - Space in between malloc and stack
    - Stack needs the space, keeps adding variables 'on top' as required by the code
        - Different "stack frames" for different 'levels' of the stack
        - Eg if we call a function in a function, we put another frame on the stack
    - Malloc is fluid and so needs its distance

```
int main() {
   int n, m;
   n = 5;   m = f(n);
}
int f(int x) {
   return g(x);
}
int g(int y) {
   int r = 4 * h(y);
   return r;
}
int h(int z) {
   int i, p = 1;
   for (i=1; i<=z; i++)
      p = p * i;
   return p
}
```

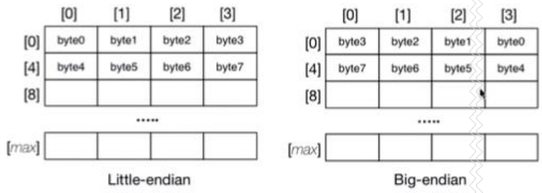


- The C view of data
    - Variables are an example of computational objects
- **Computational objects** have:
    - A **location** in memory (obtain with '&')
    - A **value** (a bit string)
    - A **name** (unless malloc'd)
    - A **type**, which determines
        - **Size** (in units of whole bytes, found with sizeof)
        - Which **options** to use to interpret its value
    - A **scope** (where it's visible in the program)
    - A **lifetime** (during which part of execution it exists)
    - Egs
        - A global variable has global region, scope of the whole file, lifetime of all runtime



            ▪ A local variable is in the stack, has a scope in main(), and has a lifetime ending with main()
            ▪ Static variables in main have global region, scope in main() and lifetime of the whole runtime
            ▪ Variables defined *after* main() can *only* be accessed in functions
            ▪ Functions are in the code, with scope everywhere and lifetime of their execution
            ▪ Function arguments are in the stack, with scope and lifetime in f()
                • Same for local function variables

- The physical view of data – essentially a very large array of bytes
    - Called main memory, RAM, primary storage, etc
        - Indexes are memory addresses (pointers)
        - Data can be fetched in chunks of 1,2,4,8 bytes
        - The cost of fetching any byte (no matter where it is) is the same (nanoseconds)
        - The byte address for an N-byte object must be divisible by N
        - Can be volatile or non-volatile
    - **-endian**



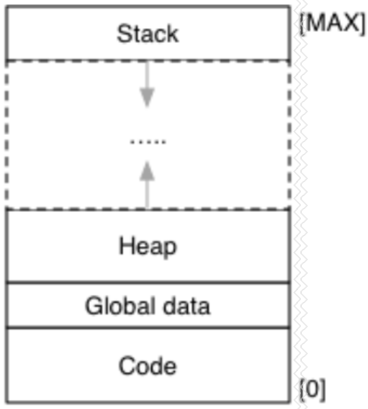
        - **Little-endian**
            - Least significant byte first, then second least second, etc
        - **Big-endian**
            - "It's when you look at it and it looks nice." -Annie
            - Most significant byte first, then second second, etc
- Data representation
    - **ASCII (ISO 646)**
        - 7-bit values, using lower 7-bits of a byte, with the top bit 0
    - **UTF-8 (Unicode)**
        - 8-bit values, with ability to extend further
        - Can encode all human language and other symbols, including mathematical signs or emojis
        - Uses a variable length encoding (one, two, three or four bytes)
        - The first byte begins with a code instructing the compiler to detect ASCII
            - Eg μ is U+00B5 which is 1011 0101 in binary and unicode **11**000010 **10**110101

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	0xxxxxxx	-	-	-
2	11	110xxxxx	10xxxxxx	-	-
3	16	1110xxxx	10xxxxxx	10xxxxxx	-
4	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

The 127 1-byte codes are compatible with ASCII
The 2048 2-byte codes include most Latin-script alphabets
The 65536 3-byte codes include most Asian languages
The 2097152 4-byte codes include symbols and emojis and ...

- Integers (three ways to write)
    - Signed decimal   42 (0-9)
    - Unsigned hexadecimal   0x2A (0-F)
    - Signed octal   052 (0-7)
    - Variations:
        - Unsigned int   123U (32-bit)
        - Long int   123L (64-bit)
        - Short int   123S (16-bits)
    - Easy data conversions
        - Binary → hex: look in blocks of four digits

- **Signed integers**
    - The first bit (digit) is 0 for positive, 1 for negative
        - Several representation options
    - **Signed magnitude**
        - First digit is sign, the rest are magnitude
        - Simple addition does not work
    - **Ones complement**
        - -N is formed by converting all bits in N
        - But we have two zeros: 11111111 (-0) and 00000000 (0)
    - **Twos complement**
        - -N is formed by converting all bits in N and adding 1
        - x = -(-x)
        - Addition works
        - There's a random 'overflow' bit we conveniently ignore



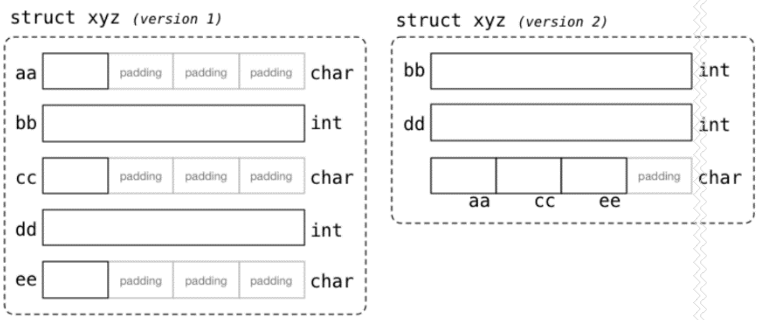
- **Different types of pointers to point to different things**
    - However they all have the same size (typically 32-bit)
    - The values must all be appropriate for the given data point
        - (char *) can reference any byte address
        - (int *) must have addr%4 == 0
- **Considering the diagram**
    - Low memory values tend to be associated with the code
    - High memory values tend to be associated with the stack
- Incrementing pointers (pointer arithmetic)
    - Moves them to the next memory address forward
    - A common (efficient) method for scanning a string is to increment a charcter pointer

- **Normalisation** – "New binary scientific notation"
    - Floats are represented by: $Num = (1 + fraction) * 2^{(exp-bias)}$
    - Exponent is represented relative to a bias value B
        - $Bias = 2^{nbits-1} - 1$
    - 1010.1011 is normalised as $1.0101011x2^{011}$
    - Always have 1 before the decimal point
        - If we had a zero we'd be wasting memory, we could just shift the whole thing
- **Floating point**
    - Float – typically 32 bit
        - 1 bit for sign
        - 8 bits for exponent
        - 23 bits for fraction
            - Each bit is $+2^{-1}$, $+2^{-2}$, etc
    - Double – typically 64 bit
        - 1 bit for sign
        - 11 bits for exponent
        - 52 bits for fraction

- Example
  - 0 10000000 11000000000000000000000
  - Sign is positive
  - Exponent is $128 - Bias = 1$
    - We thus multiply the fraction by $2^{exp} = 2^1$
  - Fraction is $\frac{1}{2} + \frac{1}{4} = 0.75$
  - Number is thus 1.75 x 2^1

- Representation
  - printf("%X.Ylf", number)
  - Prints X digits total (including the DP)
  - Y of them are after the DP

- Structs
  - *Order* of struct elements is largely determined by compiler
  - Re-ordering fields may save space



The Command Line – *A Strange Place*

- Compiling
  - gcc -E x.c (executes the C pre-processor, rewriting code)
  - gcc -S x.c (produces a file x.s containing assembly code)
  - gcc -c x.c (produces a file x.o containing relocatable machine code)
  - gcc x.c (produces a.out)
  - gcc *.o (compiles everything)
- Makefiles
  - Sometimes you don't want to compile every file, but keeping track of which files you want to compile is tedious
  - Between files:
    - Dependencies exist
      - Eg x.o depends on x.c and y.h
    - Actions state how to produce *targets* from *sources*
    - Rules combine dependencies and actions
      - Eg if x.c changes, rebuild x.o
  - **Makefiles** contain definitions and rules for compilation
    - **Target**
    - **Source**
      - If you change one of these, the blue and red files will recompile upon 'make'
    - **Action**
    - Other arguments
      - CC = compiler

```
bm : bm.o Stack.o
        gcc -o bm bm.o Stack.o

bm.o : bm.c Stack.h
        gcc -c -Wall -Werror bm.c

Stack.o : Stack.c Stack.h
        gcc -c -Wall -Werror Stack.c
```

        - CFLAGS = (compiler flags)
            - -Wall (display all warnings)
            - -Werror (display all errors)
- Commands
    - cat file
        - Prints the contents of the file
    - man some_command
        - Opens the manual page for that command

MIPS – *Man, I Program Sowell*

- A well-known and relatively simple architecture
    - Silicon graphics, NEC, Nintendo64, etc
- **MIPS features**
    - 32 x 32-bit general purpose registers
    - 16 x 64-bit double precision registers
    - PC – 32-bit register
    - HI, LO – store the results of multiplication and division
        - Two 32-bit registers to store big integers
- **Data types**
    - Byte (8 bits)
    - Halfword (2 bytes)
    - Word (4 bytes)
    - Characters require 1 byte of storage
    - Integers require 1 word of storage
- Reference
    - Registers can be referred to as $0,…,$31 or by symbolic names

Register	**Name**	**Notes**	**Cont**
$0	zero	The value 0; unchangeable	
$1	$at	Assembler temporary; reserved for assembler use	
$2	$v0	Value from expression evaluation or function return	This is what we use for all syscalls. If we load an integer it goes here. Also used for function returns.
$3	$v1	Value from expression evaluation or function return	
$4	$a0	First argument to a function/subroutine, if needed	This mean when we call a function, it immediately looks to the $a? registers
$5	$a1	Second …	
$6	$a2	Third …	
$7	$a3	Fourth …	

$8,…,$15	$t0,…,$t7	Temporary; must be saved by caller to subroutine; subroutine can overwrite	
$16,…,$23	$s0,…,$s7	Safe function variable; must not be overwritten by called subroutine	We modify these within functions
$24, $25	$t8, $t9	Temporary; must be saved by caller to subroutine; subroutine can overwrite	
$26, $27	$k0, $k1	For kernel use; may change unexpectedly	
$28	$gp	Global pointer	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Return address of most recent caller	We store the pointer to the place in our program whence we jumped.

- There are also floating point registers

- Features of MIPS language
  - # comments
  - Labels appended with '**:**'
  - Directives – a symbol beginning with '**.**'
    - **Directives list:** http://students.cs.tamu.edu/tanzir/csce350/reference/assembler_dir.html
  - Assembly language instructions

- **Pre-main tags:**
  - .data
    - .data on a line on its own introduces the data section
    - Variables can now be introduced
      - a: .word 42
      - array: .space 20
      - msg: .asciiz "Hello world"
      - Etc
  - .globl
    - .globl main declares the main function
  - .text
    - .text on a line on its own introduces the text section
    - Functions can now be introduced (beginning with their own tag)
      - main:

- General code structure and example code

```
# Prog.s ... comment giving description of function
# Author ...

     .data      # variable declarations follow this line
                # ...

     .text      # instructions follow this line
     .globl main
main:           # indicates start of code
                # (i.e. first user instruction to execute)
                # ...

# End of program; leave a blank line to make SPIM happy
```

```
    .data
a:  .word 42                  # int a = 42;
b:  .space 4                  # int b;
    .text
    .globl main
main:
    lw   $t0, a               # reg[t0] = a
    li   $t1, 8               # reg[t1] = 8
    add  $t0, $t0, $t1        # reg[t0] = reg[t0]+reg[t1]
    li   $t2, 666             # reg[t2] = 666
    mult $t0, $t2             # (Lo,Hi) = reg[t0]*reg[t2]
    mflo $t0                  # reg[t0] = Lo
    sw   $t0, b               # b = reg[t0]
    ....
```

- Instructions (all 32-bit)
    - **Instructions list:** http://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf
    - lw – load word (assign value from a memory (word) address)
    - li – load integer (assign value)
    - la – load address
    - add – add (requires three variables)
    - mult – multiply (requires two variables and stored in Lo,Hi)
    - mflo / mfhi – store the value from Lo / Hi in a specified variable (must call one at a time)
    - sw – store word (store a value into a memory (word) address)
    - mov – move a value from one register to another
- Memory layout

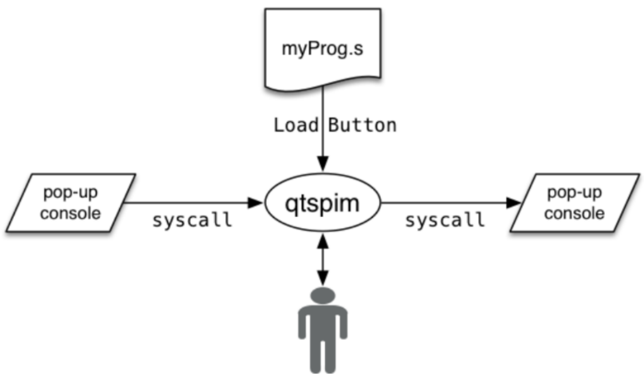
Region	Address	Notes
Text	0x00400000	Contains only instructions (read only)
Data	0x10000000	Data objects (read/write)
Stack	0x7fffefff	Grows down (read/write)
k_text	0x80000000	Kernel code. Only accessible kernel mode (read only)
k_data	0x90000000	Kernel data. Only accessible kernel mode (read only)

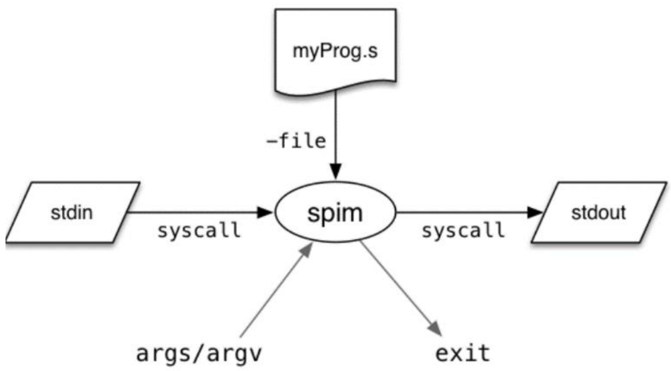
- **Classes of instructions**
    - Load and store
        - Transfer data between registers and memory
    - Computational
        - Performs arithmetic/logical operations
    - Jump and branch
        - Transfer control of program execution
    - Coprocessor
        - Standard interface to various co-processors
    - Special
        - Miscellaneous tasks

- **syscall**
    - A special instruction
    - Several types of calls
    - Called from a $v register
        - li $v0, 5
          syscall
        - The above scans a value into $v0

Service	Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = char *	
read_int	5		integer in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0 = buffer, $a1 = length	string in buffer (including "\n\0")

- SPIM instructions:





https://cgi.cse.unsw.edu.au/~cs1521/19T2/docs/spim.php
    - SPIM has its own useful instructions that are mapped to one of more MIPS instructions
- Directive note
    - Directives apply to everything below the directive, before a new directive
- The importance of .align
    - .align n means we jump to the nearest specified block of memory dividible by 2^n
    - This is critical for storing certain data types

- Addressing modes
    - lw $t0, var (address via naming)
    - lw $t0, ($s0) (indirect addressing)
    - lw $t0, 4($s0) (indexed addressing – accesses from the fourth element/byte)

```
lw   $t1,label       # reg[t1] = memory[&label]
sw   $t3,label       # memory[&label] = reg[t3]
                     #   &label must be 4-byte aligned
la   $t1,label       # reg[t1] = &label
lui  $t2,const       # reg[t2] = const << 16
and  $t0,$t1,$t2     # reg[t0] = reg[t1] & reg[t2]
add  $t0,$t1,$t2     # reg[t0] = reg[t1] + reg[t2]
                     #   add as signed (2's complement) ints
addi $t2,$t3, 5      # reg[t2] = reg[t3] + 5
                     #   "add immediate" (no sub immediate)
mult $t3,$t4         # (Hi,Lo) = reg[t3] * reg[t4]
                     #   store 64-bit result in registers Hi,Lo
seq  $t7,$t1,$t2     # reg[t7] = (reg[t1]==reg[t2]) ? 1 : 0
j    label           # PC = &label
beq  $t1,$t2,label   # PC = &label if (reg[t1] == reg[t2])
nop                  # do nothing
```

```
    .text        # following instructions placed in text
    .data        # following objects placed in data

    .globl       # make symbol available globally

a:  .space 18    # uchar a[18];  or  uint a[4];
    .align 2     # align next object on 2^2-byte addr

i:  .word 2      # unsigned int i = 2;
v:  .word 1,3,5  # unsigned int v[3] = {1,3,5};
h:  .half 2,4,6  # unsigned short h[3] = {2,4,6};
b:  .byte 1,2,3  # unsigned char b[3] = {1,2,3};
f:  .float 3.14  # float f = 3.14;

s:  .asciiz "abc"
                 # char s[4] {'a','b','c','\0'};
t:  .ascii "abc"
                 # char s[3] {'a','b','c'};
```

            - Like pointer arithmetic, but this only does 4 bytes (*not* 4*4 bytes)
- **Addresses** are hard :((a and b are registers and/or variables)
    - la a, b
        - Takes the ADDRESS of b and puts that in a
    - lw a, b
        - Takes the CONTENT of b and puts that in a
    - li a, num
        - Puts the number num into a
    - lw a, (b)
        - a = *b
    - lw a, 4(b)
        - a = *b + 4
- **Additional loading**
    - lb load one byte (lbu for unsigned)
    - lh load two bytes (lhu for unsigned)
    - lw load four words
    - la load the address

- **While loops**

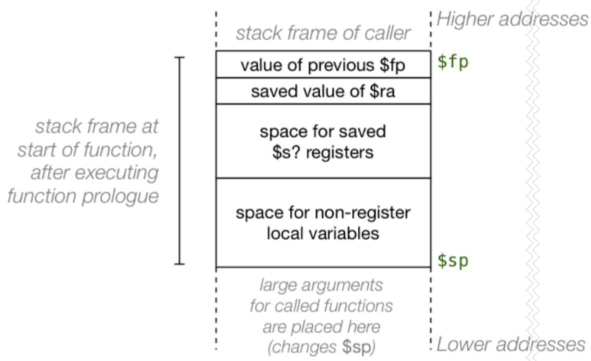
```
while ($s1 < $s0) {
    ...
}


while:
    bge $s1, $s0, end_while
    ...
    addi $s1, $s1, 1

end_while:
```

<u>More C to MIPS:</u>

```
                    int x = 5;          x: .word 5
                    int y;              y: .space 4

z = 5 * (x+y);      t = x + y;          lw $t0, x
                                        lw $t1, y
                                        add $t0, $t0, $t1

                    t = 5 * t;          li $t1, 5
                                        mul $t0, $t0, $t1

                    z = t;              sw $t0, z

                    x = 2;              li $t0, 2
                                        sw $t0, x
```

```
i = 0; n = 0;        i = ; n = 0;
while (i < 5) {      loop:
    n = n + i;           if (i >= 5) goto end;
    i++;                 n = n + i;
}                        i++;
                         goto loop:
                     end:


if (cond)            if_stat:                        if_stat:
    {statements1}        t0 = (cond)                     t0 = evaluate (cond)
[not actual MIPS]
else                     if (t0 == 0)                    beqz $t0, else_part
    {statements}             goto else_part              statements1
                         statements1                     j end_if
                         goto end_if                 else_part:
                     else_part:                          statements2
                         statements2                 end_if:
                     end_if:
```

- **Functions**
    - The process
        - The arguments are evaluated and set up for function
        - Control is transferred to the code for the function
        - Local variables are created
        - Function code executes
        - The return value is set up
        - Control transfers back to where the function was called from
        - The caller receives the return value
    - Stack frames
        - Each function allocates a stack frame
        - Uses:
            - Saved registers
            - Local variables
            - Parameters to callees
        - Created in the *prologue* (push)
        - Removed in the *epilogue* (pop)



-
    - Conventions
        - Caller saved registers ($t0…$t9, $a0…$a3, $ra)
            - f() calls g() knowing these registers may change
        - Callee saved registers ($s0…$s7, $sp, $fp)
            - f() calls g() assuming the registers will be unchanged
            - If g() modifies them, they will first be saved and then restored
    - Call with instruction “jal func”
    - Put return value in $v0

- End instruction is “jr $ra”
    - This jumps back to where we left off with jal

```
    # Set up stack frame.
    sw  $fp, -4($sp)     # push $fp onto stack
    la  $fp, -4($sp)     # set up $fp for this function
                         # Using $sp and $fp we chuck all our
                         #  variables onto the stack
    sw  $ra, -4($fp)     # save return address
    sw  $s0, -8($fp)     # save $s0 to use as ...
    sw  $s1, -12($fp)    # save $s1 to use as ...
    sw  $s2, -16($fp)    # save $s2 to use as ...
    addi    $sp, $sp, -20   # move $sp to top of stack


# Alternatively:

addi $sp, $sp, -4
sw $fp, ($sp)
move $fp, $sp
addi $sp, $sp, -4
sw $ra, ($sp)
addi $sp, $sp, -4
sw $s0, ($sp)
addi $sp, $sp, -4
sw $s1, ($sp)
addi $sp, $sp, -4
sw $s2, ($sp)


    # clean up stack frame
                         # We retrive all our variables
                         #  from the stack
    lw  $s2, -16($fp)    # restore $s2 value
    lw  $s1, -12($fp)    # restore $s1 value
    lw  $s0, -8($fp)     # restore $s0 value
    lw  $ra, -4($fp)     # restore $ra for return
    la  $sp, 4($fp) # restore $sp (remove stack frame)
    lw  $fp, ($fp)  # restore $fp (remove stack frame)

    jr $ra
```

- **Function calling protocol**
    - Before calling
        - Place arguments in $a0…$a3
        - If more than 4 args, push all args onto the stack

- Save any non-$s registers that need to be preserved by pushing onto the stack
- jal address of the function (usually labelled)

```
# start of function
FuncName:
   # function prologue
   #   set up stack frame ($fp, $sp)
   #   save relevant registers (incl. $ra)
   ...
   # function body
   #   perform computation using $a0, etc.
   #   leaving result in $v0
   ...
   # function epilogue
   #   restore saved registers (esp. $ra)
   #   clean up stack frame ($fp, $sp)
   jr  $ra
```

- Pushing onto the stack (eg $t0)
  - addi $sp, $sp, -4
    sw $t0, ($sp)
  - OR
  - sw $t0, -4($sp)
    addi $sp, $sp, -4
-

- Arrays
  - a: .space 20 (holds 5 elements, ie 20 bytes worth of words)
  - a: .word 1, 3, 5, 7, 9 (a = {1,3,5,7,9})
  - **How do we access the numbers?**
  - *Increment the memory location by 4 (if we are dealing with words)!*

```
int sum, i;                sum: .word 4         # use reg for i
int a[5] = {1,3,5,7,9};    a:   .word 1,3,5,7,9
...                             ...
sum = 0;                        li   $t0, 0    # i = 0
                                li   $t1, 0    # sum = 0
                                li   $t2, 4    # max index
for (i = 0; i < N; i++)    for: bgt  $t0, $t2, end_for
                                move $t3, $t0
                                mul  $t3, $t3, 4
   sum += a[i];                 add  $t1, $t1, a($t3)
printf("%d",sum);               addi $t0, $t0, 1     # i++
                                j    for
                       end_for: sw   $t1, sum
                                move $a0, $t1
                                li   $v0, 1
                                syscall          # printf
```

  - Note: a($t3) is &a + $t3

- 2D Arrays
  - To access array[row][col]
  - We access offset(array), where offset = row*rowsize + col
  - Make sure to multiply this offset by 4 if we are dealing with words!

- **Structs**
  - Literally just define space
  - Stu1: .space 56
  - You then put values into the struct according to offsets

    - You save pointers to structs if you want to work with them
- **argc and argv**
    - These are just put on main's stack frame

- **Data structures**
    - char as one byte in memory
    - int as one word in memory
    - double as two words in memory
        - All of the above could be implemented:
            - In a register (small scope)
            - On the stack
            - In .data if we need longer lifetimes (but don't do this lol)
    - arrays sequence of memory bytes/words
    - structs chunk of memory accessed by offsets
    - linked structures struct containing address of another struct
- Static VS dynamic
    - Static:
        - Uninitialized memory is allocated at compile time
            - val: .space 4
        - Initialised memory allocated at compile time
            - val: .word 5
    - Dynamic:
        - Put things in registers