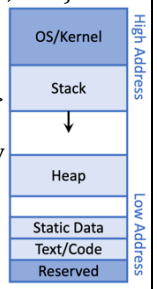
Lecture 1: Introduction

Ideas: abstraction, locality, parallelism (pipelining/multi-core), Redundancy (dependability)
Abstraction: High-level language program(*.c) ->(Compiler) Assembly program(*.s) (RISC-V) ->(Assembler) Machine object(*.o) ->(Linker) Machine code (executable) -> Instruction set architecture <-(Implement) Microarchitecture
Assembly program: text representation of machine code; Machine code: binary sequence encodes program instructions. Parallelism: Speedup = $1/(1-p+p/n)$
Aligned address: multiple of word size (pointer size)
Unsigned: zero extension (logical right shift) Signed: sign extension (arithmetic right shift).
Mix: signed value cast to unsigned value; overflow when truncated bit different from MSB of result
Floating point: 1 bit sign, 8/11 bit exponent, 23/52-bit frac, $v = (-1)^s \times 2^{e-127} \times 1.f$
Multi-Dimensional Array: `&a[i][j]==a+(i*C+j)`, continuous
Multi-Level Arrays: several memory accesses (array of pointers), not continuous
Structures: within structure, each field align to its size, overall structure, ordered
Unions: share same memory, size is largest field size both align to largest alignment requirement
Big Endian: MSB in lower address (network) Little Endian: MSB in higher address (x86, ARM)
Interpretation: first convert into system-independent bytecode -> machine code (java)
just in time (JIT) compilation: compile and execute on the fly at runtime.



Lecture 2: Processors and Assembly Code

Von Neumann Architecture: Input / Output <-> Processor (Control Unit, Compute Unit) <-> Memory (separate processor and memory)
Von Neumann: a single memory; Harvard: separate memory for data and program memory
ISA: Hardware-software interface, define states (memory, register, stack) and instructions (functionality, format)
two processor types: General-purpose/programmable processor (flexibility) and Specialized/customized accelerator (efficiency)
State: PC (address of next instruction), memory, register (decided by compilers or assemble programmers); ALU (arithmetic logic unit); Controller: parse instruction, decide next PC
Instruction Execution: Fetch (Update PC), Decode, Execute, Load/store, Write back.
Optimization: Pipelining, Superscalar (dynamic find independent instructions), Multicore (PC/ register private, memory shared), Specialized/Customized Hardware: high efficiency but inflexible
RISC: risc-v, arm, each instruction specify single operation, save energy and power
CISC: x86, internally translate code into RISC-style, early processor

char	1	1
short	2	2
int	4	4
long	4	8
float	4	4
double	8	8
Pointers	4	8

RISC-V ISA: PC/register(x0-x31) 32-bit, x0 always 0, all RISC-V instructions are 32-bit long.

```
<op> rd, rs1, rs2        # input rs1, rs2, output rd
<op>i rd, rs1, imm       # op includes add, sub, xor…
slli rd, rs1, imm        # shift left logical immediate
lw rd, offset(rs1)       # load word
sw rs2, offset(rs1)      # store word                  j L # jump to L (unconditional)
beq rs1, rs2, L          # branch to L if rs1 == rs2   (bne, blt, bge)
```

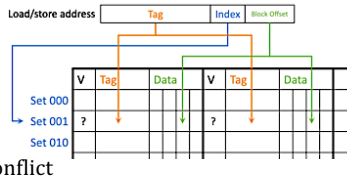
multiply size of element when access array ISA support for Parallelism: SIMD
ISA Support for Sync: Atomic instructions (coordinate among multiple processors)
`t&s reg, addr`, usage: `reg = *addr, if (reg == 0) *addr = 1`, build a spin lock

```
lock: t&s reg, addr          unlock: st addr, 0
      bne reg, x0, lock /* if not 0, try again */
```

Text/Code: set at loading, stored in program binary file; Static Data: global; Heap: dynamic allocate.
Stack: support procedure call (arguments, local/temporary variables); LIFO match 嵌套过程调用
Frame: space for procedure 实例化; caller or callee save registers before call, restore after call
Procedure Calls: 1. Passing control: jump from caller to callee; Passing data: procedure arguments and return values (on register, if too much put on stack); allocate variable; save/restore registers
`jr ra` jump to return address in `ra`
`jal L` jump-and-link, jump to L and save return position to register `ra(x1)`

Lecture 3: Memory Hierarchy

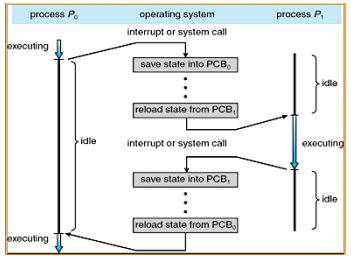
Primary storage: main memory, DRAM (1T1C, separate chips, destructive cell access), optimized by SRAM (6T, on the same chip) caches Secondary storage: disk/SSD, external, managed by file system in OS
Reason: want fast access but not persist during execution (faster but volatile); permanent data need stronger persistency and consistency; program has private address space but share data
Tradeoff: large memory capacity vs. fast memory access speed; larger performance gap between processor and memory Temporal locality: reference scalar variable, loops
Spatial locality: reference array elements, fetch instruction in sequence; pointer chasing lose locality.
Memory Hierarchy: faster device at level k store a subset of data for slower device at level $k+1$
Pyramid: On-chip: Register, L1, L2, L3(SRAM); Off-chip: Main memory (DRAM), NVM, Secondary storage (SSD, Disk); Remote Data in cache transfer in unit of blocks, a.k.a cachelines
Hit latency = time to access cache Miss penalty = overhead of fetching data from memory on a miss = time to access memory + time to deliver to cache + (time to replace in cache)
AMAT = hit latency + miss rate * miss penalty (tradeoff between hit latency and miss rate)
Fully associative(N=#blocks): any block can be placed in any cache line; Low miss rate, high hit latency
Direct mapped ($N=1$): each block has only one place to go; low hit latency, high miss rate; index = block address % blocks in cache block address = address / block size
(N-way) Set associative: a block can go to one of N locations(set); set index = block address % sets, all locations in a set must be searched
cache entry = data block + tag + valid bit + more metadata
Block size -> Compulsory miss: first access to a block.
Capacity -> Capacity miss: cache is too small.
Associativity -> Conflict miss: item was replaced because of location conflict



Replacement policy: determine whether/which to evict; Random, LRU, LFU
whether to evict: normal (replace an existing old block), bypass (don't enter cache)
write-back + write allocate: write-hit: write data to cache, defer write to memory until eviction (indicate by a dirty bit) write-miss: treat as read-miss, then writes to cached block
Multi-Level Caches: Miss penalty = AMAT of the next level; Miss rate = local miss rate
L1 attach to core, split D-cache & I-cache; L2 unified cache per core; L3 shared among cores.
Coherence Protocols: Update-based: a write occurs, propagate new value to other private caches; Invalidate-based(modern): a write occurs, invalidate other copies. Depend on data sharing characteristics. Processor caches managed by hardware, cache config not necessarily part of ISA, functionally transparent to software.
cache performance optimizes not portable across processors, performance not transparent to software.

Lecture 4: Processes and Threads

Process: an instance of a executing program; Execution context: have exclusive use of processor (by context switch); Address space: have exclusive use of memory (by virtual memory)
Thread: a single unique execution context (registers, PC, stack pointer, memory), a process can have multiple threads, thread can be executing or suspended
a process = one or more threads (concurrency) + address space (purpose: protection)
Context switch: between threads, swapping processor states (register, PC, stack pointer), data remain in memory, coordinated by OS.
Multiprocessing: time-interleaved, address space all mapped to memory, States (PC, register) save to memory before descheduled, restore before continue
Address space: seem to have exclusive use of memory, main purpose: protection, different processes can't freely access each other's data memory content/ address space shared by all threads of a process, each thread has private register, PC, stack pointer (stack not shared)



OS kernel represents a process as process control block (PCB), an in-memory data structure contains process states.
Hardware support one bit (user/kernel mode bit), certain privileged instructions only available in kernel mode
User to kernel transition (save user process PC): syscall (process exit, allocate memory, send data to network): proactive, save and restore PC/register, offer a narrow waist as a portable interface; interrupt (Ctrl-C, timer): reactive, external; exception and trap (segfault): reactive, internal.
OS handle exceptions/interrupts: save current PC and cause, transfer control flow to the kernel; exceptions are processed by kernel exception handler (selected by cause); return to user process after handling or abort.
Process is terminated if returning from main function, exit syscall or receiving a signal (Default action is terminate) `void exit(int status);` never returns to the program, non-zero status error
Parent process creates a new running child process by calling `fork`; almost identical to parent process, new PID, identical (but separate) copy of address space, identical copy permission of parent I/O; execute concurrently.

`int fork(void)` return 0 to child process, return child PID to parent process, return -1 if failed; called once but returns twice.
a program terminates but still occupies system resources, called zombie process, e.g. parent not terminate, child exit; parent process should call `wait` or `waitpid` to clean up zombie process.
`pid_t wait(int *wstatus);` block until one of its children terminates, return child PID, status is set to child exit status if `wstatus != NULL`
`pid_t waitpid(pid_t pid, int *status, int options);` wait for a specific child process
Parent terminates without reaping a child, orphaned child is reaped by `init` process, must explicitly kill child, i.e. child process is still active though parent has terminated
`int execve(char *filename, char *argv[], char *envp[]);` retain PID, overwrite code, data and stack; called once and never returns; fork-then-exec, parent wait for child to exec
Signal: small message to notify a process that event has occurred; sent from kernel, identify by signal ID; Sending a signal: kernel detects a system event / process calls `int kill(pid_t pid, int sig);` Receiving a signal: Terminate process, execute signal handler, ignore signal.
Shell: a process that fork itself, exec program, wait; Foreground: shell waits and reaps; Background: not reap by shell/init, use signal to manage or bring to foreground
Scheduling: wait in ready queue; preemptive: OS decide when to switch; non-preemptive: once giving some time to execute can't take processor back before period ends
Respone time = wait time + exec time; Throughput = number of processes completed per unit time
Convoy effect: short jobs are blocked behind long jobs.
Scheduling policy: FIFO(FCFS), RR, SJF/SRTF (optimal policy to minimize average response time)

Lecture 5: Practical Skills in Computer Systems

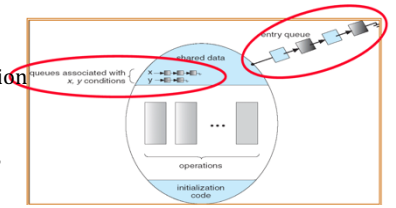
Shell: Interpreted language, original string + list and interpret, define inter-process communication
Basic shell commands: `date`: date and time, `time`: how long a command runs, `uptime`: time system, `uname`: print system information, `clear`, `man`, `whoami` `ls`, `cd`, `mkdir`, `pwd`, `cat`: concatenate and print, `less/more`, `head`, `tail`, `tail -f`: follow the growth of a log file, `df -kh`: disk space usage in kB, `wc -l`: count lines, `ls -al`: list include hidden file keyboard + screen = terminal
Permissions: d(rwx)*3, owner, group, other
stdin:0 -> process -> stdout:1, stderr:2, three open files on the start of process; process inherit all open files, env variables from parent. check by env, set by env variable `PS1`
`grep t > ./out 2> ./error` redirect stdout(contain t) to `./out`, stderr to `./error`
`&>` redirect stdout and stderr, `2>&1` redirect stderr to stdout, `tee` redirect to both stdout and file
useful terminal shortcuts: `ctrl+C` = `SIGINT`, `ctrl+\`=`SIGQUIT`, `ctrl+Z` = `SIGSTP` (stop and background), `ctrl+D` = EOF, `ctrl+R` = search history, `ctrl+A/E` = start/end of line
Process management: `ps`: list all processes in shell; `ps -ef` for all processes in system; `top` for real-time process monitoring; `kill`: send any signals, `bg` and `fg`: `ctrl+Z` to stop a process, then `bg` to run in background, `fg` to run in foreground, `jobs` to list all jobs in the shell;
Background job: can't output to terminal, blocks when trying to write to terminal, need to redirect.
Shell globbing: `convert image.{png,jpg};` `cp /path/{foo,bar}.sh /newpath`
`mv *{.py,.sh} new_folder;` `mkdir foo bar && touch {foo,bar}/{a...h}`
regular expression: `.` any single character, * 0 or more preceding, + 1 or more preceding, ? 0 or 1, ^ start of line, $ end of line, `a{3}` 3 a's, `a{3,}` 3 or more a's, `a{3,5}` 3 to 5 a's, `[abc]` match any character in brackets, `[^abc]` match any character not in brackets, `(RX1|RX2)` matches RX1 or RX2, `\d` digit, `\w` word character(a-z, A-Z, 0-9, _), `\s` whitespace, `(?!mail)` stops if match mail, otherwise ignore.
Always use \ to escape special characters!
`\d+(\.\d\d)?` 114, 114.514 `[2-9]|[12]\d|3[0-6]` match 2-36
`((\d{3})(\.|-))?(\d{3})(\.|-)(\d{4})` 114.514.1919, 114-514-1919, 666-7777
`sort -n`: use numeric order, `-k`: on key, `sort -nk1,1` sort on first column; `uniq`: remove duplicate lines, only work on sorted data; `ssh`: `cat localfile | ssh user@host tee serverfile`

Lecture 6: Concurrency and Synchronization (1)

Cooperating threads: Share resources, speedup, Modularize
Communication mechanism: Separate address space isolate process; two kinds: Shared-Memory Mapping, Message Passing -- `send()` `receive()`, works across network
Web server with Thread Pool:

```
master() { allocThreads(slave,queue);      slave(queue) { while(TRUE) {
    while(TRUE) {                              con=Dequeue(queue);
        con=AcceptCon();                       if (con==NULL) sleepOn(queue);
         Enqueue(queue,con);wakeUp(queue); }}  else ServiceWebPage(con); }}
```

Sync: using atomic instructions to ensure cooperation
Critical Section: code that only one thread can execute at a time
Mutual Exclusion: ensure only one thread executes in critical section
Fine grain sharing: increase concurrency, more complex.
Coarse grain sharing: decrease concurrency, less complex (lock)
Lock: lock before entering critical section / accessing shared data, unlock when leaving, wait if locked (all sync involves waiting)




```
Thread A() {                              Thread B() {
    leave note A;                             leave note B;
    While (note B) { do nothing; }            if (noNote A) {
    if (noMilk) { buy milk; }                 if (noMilk) { buy milk; } }
    remove note A;                            remove note B;
}                                         }
```

`Lock.Acquire()` wait until lock is free, then grab. `Lock.Release()` unlock, waking up anyone waiting.
Semaphore: a generalization of lock, has a non-negative integer value (`P` waits until value > 0, then decrement; `V` increment value, wake up waiting), can't wait inside critical section
Mutual Exclusion (init 1): `P; critical section; V;`
Scheduling Constraints (init 0): `ThreadJoin: P; ThreadFinish: V;`
Producer-Consumer Problem: set three Semaphores, empty, full, mutex, intial value of empty = buffer size, full = 0, mutex = 1 the code between `mutex.P()` and `mutex.V()` is critical section

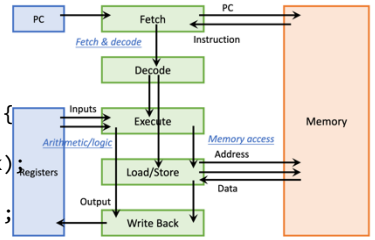
```
Producer() {                              Consumer() {
    empty.P(); mutex.P();                     full.P(); mutex.P();
    Enqueue(); mutex.V(); full.V();           Dequeue(); mutex.V(); empty.V();
}                                         }
```

order of P matters, otherwise it cause deadlock; V can be in any order.
Monitor: a lock (mutual exclusion to shared data) and zero or more condition variables for managing concurrent access to shared data.
condition variable: a queue of threads waiting for sth inside critical section; Key idea: allow sleeping inside critical section by atomically releasing lock.
ops (must hold lock when calling): `Wait(&lock)` release lock, sleep, reacquire lock when wake up `Signal()` wake up one thread in the queue `Broadcast()` wake up all threads in the queue
Hoare Monitor: signaler give lock and CPU to waiter, signaler wait until waiter exits critical section or waits again (if) Mesa Monitor: signaler keep lock and cpu, waiter placed on a local "e" queue for monitor, need to check condition again after waking up (while)

Lecture 7: Concurrency and Synchronization (2)

Reader-Writer Problem:

```
Reader() {                                Writer() {
lock.Acquire()                            lock.Acquire();
while (AW + WW > 0) {                     while (AR + AW > 0) {
    WR++;                                 WW++;
    okToRead.Wait(&lock);                 okToWrite.Wait(&lock);
    WR--; }                               WW--; }
AR++; lock.Release();                     AW++; lock.Release();
// read                                   // write
lock.Acquire(); AR--;                     lock.Acquire(); AW--;
if (AR == 0 && WW > 0) #optional          if (WW > 0) okToWrite.Signal();
    okToWrite.Signal();                   elif (WR > 0) okToRead.Broadcast();
lock.Release(); }                         lock.Release(); }
```



Note: if only use one condition variable, change `Signal` into `Broadcast`
Performance: User -> Kernel -> Acquire -> User (op on critical section) -> Kernel -> Release -> User
C-support for Sync: release lock when return. Go-support for Sync: use communication to share instead of memory Java-support for Sync: `synchronized` keyword, `notify` and `wait`
C++-support for Sync: Consider exited path caused by exceptions. Must catch all exceptions in critical sections or Use lock class destructor to release.
Resources: Preemptable: CPU, can take away; Non: disk space, critical section, must leave with thread.
Deadlock: mutual exclusion, hold and wait, no preemption (only released by the thread holding resource), circular wait. Starvation: thread waits indefinitely
Techniques for preventing deadlocks: don't allow waiting; make threads request everything they need at beginning (over-estimate); Force all threads to request all locks in a particular order.