Computer Systems

1 Representing Information

- Know conversions between binary, hexadecimal, and decimal.
- Word size w, size of addressable memory 2^w bytes.
- Little endian: *l.s.b.* first. Big endian: *m.s.b.* first.
- Boolean Algebra:
  - Bitwise operators: `&, |, ^, ~, <<, >>` (Right shift is arithmetic, left shift is logical)
  - Logical operators: `&&, ||, !`
- Operator Precedence:
  1. `++`, `--` (suffix), function calls, array access, struct access.
  2. `++`, `--` (prefix), `!`, ~, `*` (dereferencing), `&` (address of), typecasting.
  3. `*,÷, %`
  4. `+, -`
  5. `<<, >>`
  6. `<, >, <=, >=`
  7. `==, !=`
  8. `&` (bitwise)
  9. `^` (bitwise)
  10. `|` (bitwise)
  11. `&&`
  12. `||`
  13. `?:` (conditional)
  14. `=, +=, -=`, etc. (assignment operators)
- *Unsigned.* Range: $[0, 2^w]$.
  - Addition:
$$x +_w^u y = \begin{cases} x + y, & \text{if } x + y < 2^w \\ x + y - 2^w & \text{otherwise} \end{cases}$$
  - Negation
$$-_w^u x = \begin{cases} x, & \text{if } x = 0 \\ 2^w - x & \text{otherwise} \end{cases}$$
  - Multiplication: $x *_w^u y = (x \cdot y) \bmod 2^w$.
- *Two's Complement.* Range: $[-2^{w-1}, 2^{w-1} - 1]$.
  - Schema:
$$[x_1 x_2 \dots x_w] := -x_1 \cdot 2^{w-1} + \sum_{i=2}^{w} x_i 2^{w-i}$$
  - Addition/Multiplication: Overflow is truncated. Negation works normally, except $-T_{min} = T_{min}$.

1.1 Floating Point

- `float` $(1, 8, 23)$ and `double` $(1, 11, 52)$. Bias $b = 2^{k-1} - 1$, where k is the size of the `exp` field.
  - If `exp` is all ones and
    - `frac` is all zeros, then $\pm\infty$.
    - `frac` is nonzero, then `NaN`.
  - If `exp` is all zeros, then $E = 1 - b$, $M = 0.[\texttt{frac}]$, and $N = (-1)^s \cdot M \cdot 2^E$.
  - Otherwise, $E = \texttt{exp} - b$, $M = 1.[\texttt{frac}]$, and $N = (-1)^s \cdot M \cdot 2^E$.

Arithmetic is performed precisely, then values are rounded according to the current rounding mode. The default rounding mode: Round to nearest representable number; if equally near, then round to the nearest even.

1.2 Conversions

- `int` $\leftrightarrow$ `unsigned`: Fix the bit stream, but reinterpret value according to new schema.
- `float/double` $\rightarrow$ `int`: Round to zero.
- `int` $\rightarrow$ `short`: Truncate and reinterpret.
- `short` $\rightarrow$ `int`: Sign extend if signed, zero extend otherwise.
- Comparisons:
  - If any argument is a `double`, then the other will be converted to a `double`.
  - Otherwise, if any argument is a `float`, then the other will be converted to a `float`.
  - If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
  - If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
  - Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

2 Assembly Programs

- Any primitive type of k bytes, should have an address that is a multiple of k.
  - For structs, each elements should satisfy its alignment requirement (requires internal padding). Every struct should have an alignment of K where K is the largest type of any of its elements. Thus, struct length and starting address should be a multiples of K (end padding).
- Buffer overflow protection
  - *Randomized stack.* Call `alloca` to randomize stack position at start of program. Use `%rbp` as base pointer to manage, when `alloca` is used.
  - *Stack corruption detection.* Use a randomized canary value in the stack, to be inserted between local buffer and the rest of the stack.
  - *Non-executable code segments.* Stack memory can be flagged as non-executable.

- Memory addressing modes, `%rax` is return register, `%rsp` is stack pointer, array pointer arithmetic, argument registers: `%rdi, %rsi, %rdx, %rcx, %r8, %r9`.
- Stack address grows downwards. Callee must push callee-saved register values onto stack (only if it plans to use them). Local variables are pushed onto the stack. If $n > 6$ arguments are used in a call, they are pushed onto the stack (right to left).

3 Processor Architecture

- `MUX` (multiplexer): Given inputs a, b and signal s, returns $sa \vee \overline{s}b$.
- `EQ` (bit equality): Given inputs a, b, returns $ab \vee \overline{a}\overline{b}$.
- *Decoder.* Given inputs a_0, a_1, outputs $d_0 = \overline{a}_0\overline{a}_1$, $d_1 = \overline{a}_0 a_1$, $d_2 = a_0\overline{a}_1$, $d_3 = a_0 a_1$. (Used to decide which register is written). A *multiplexer* decides which registers are read.
- A multiplexer also decides which operation is performed by ALU.
- *Sequential logic.* Computation with storage (e.g. DFA). *Combinational logic.* Output is only a function of present inputs.

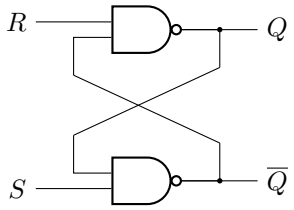


Figure 1: RS Latch.

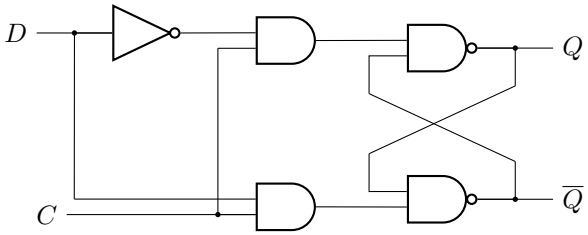


Figure 2: D-Latch (Flip-flop) circuit.

- *Fetch.* Reads bytes in memory according to PC. May fetch register bytes (for register operands). May fetch constants. Computes next address of next instruction (sequentially).
- *Decode.* Reads up to two operands from register file.
- *Execute.* ALU performs operation indicated by instruction, computes effective address of memory reference, or increments/decrements stack pointer. Condition codes are possibly set.
- *Memory.* Read and write data to memory.
- *Write back.* Write results to register file.
- *PC update.* PC is set to address of next instruction.

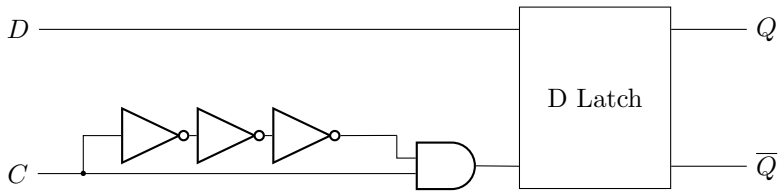
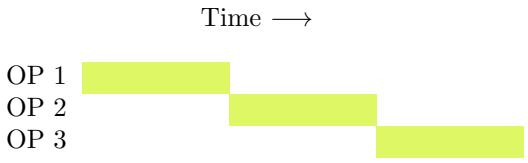


Figure 3: Edge triggered flip-flop.

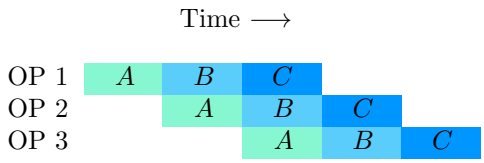
3.1 Pipelining

The total time required to perform a single instruction is known as *latency*. *Throughput* is the rate at which instructions can be processed. Nonuniform partitioning of combinational logic results in a *bottleneck*, where the logic with the greatest latency limits the clock rate. Moreover, there are diminishing returns on deep pipelining, since adding more pipeline registers results in greater overhead. Interleaving, where multiple copies of a 'slow' component are used and are interleaved cycle by cycle, can improve pipeline.



- Control dependencies occur when one instruction determines the location of the following instruction, as in jump, return, or call.
- Data Dependencies occur when results computed in one instruction are used as data in the following instruction.

In either case, we must wait until after the execution phase of the preceding instruction before, continuing with the next instruction.



- Inserting `nop` instructions can eliminate data and control dependencies, but this wastes valuable pipelining space.
- Delay slots, where non-dependent instructions are inserted between dependencies, can be used instead of `nop`. (Done by assembler). Out-of-order execution can also be achieved using hardware (see Tomasulo algorithm).
- Specialized hardware can be used to detect when dependencies occur. Stall and bubble (`nop`) flags, can then be sent to the appropriate pipeline register.
- The most common technique to resolve control dependencies is (dynamic) branch prediction. For example, a 1-branch predictor will take the branch, if the branch was previously taken, and not take the branch otherwise. Specialized memory in the processor called the *branch target buffer* keeps track of the target for each branch prediction. Specialized hardware ensures the

machine state isn't manipulated while executing speculative code. If a misprediction occurs, the mis-executed instructions are killed, and execution resumes from the correct target.

- For data dependencies, data forwarding by specialized hardware can send results directly from the execute and memory stages to the decode stage of the instruction that needs them. (No bubbles)

4 Compiler Optimizations

GCC allows options `-Og`, `-O1`, `-O2`, `-O3`.

Compilers can only apply optimizations that it is certain do not change the behavior of the program. It is the duty of the programmer to write programs in a way that the compiler can transform them into efficient machine-level code. For example, memory aliasing occurs when two pointers designate the same memory location. The compiler must assume different pointers may be aliased (*optimization blocker*). Another optimization blocker is function calls since they may have side effects that modify the global program state.

Note: Function calls can be optimized via inline substitution, where the function call is replaced by the body of the function.

Performance Metric: CPE or cycles per element. Measures the (amortized) number of clock cycles required to compute one element (as in an array or matrix). How to optimize:

- Repeatedly attempt different approaches
- Performing measurements, and
- Analyzing assembly code to identify bottlenecks.

Examples of loop optimizations.

- Pre-computing and storing values that remain constant. (e.g. $i <$ `length`(v) in a loop conditional)
- Eliminate function calls inside loop whenever possible.
- *Code Motion*: Hoist/lower any loop-invariant code. (Identify a computation that is performed multiple times, but the result of the computation does not change. Then move the computation where it won't be repeatedly computed.)
- *Strength Reduction*: Expensive operations are replaced with equivalent but less expensive operations. (e.g. bit-shift instead of `*`)
- *Common Subexpression Elimination*: Computing and storing a subexpression that is used multiple times in a loop.

4.1 Exploiting Microarchitecture

Latency bound. Encountered when a series of operations must be performed in strict sequence. Limits ability of processor to exploit instruction-level parallelism. *Throughout bound*. Characterizes raw computing capacity of processor. This is the ultimate limit of program performance.

- *Loop Unrolling*: Computing multiple elements per loop iteration. Reduces number of loop index calculations and conditional branching (*overhead*). Exposes ways in which code may be transformed to reduce computations along the critical path.

Note: To unroll a loop by a factor of k, set the upper limit to $n - k + 1$. Then manually 'combine' the rest.

The issue with loop $k \times 1$ unrolling is that it cannot improve beyond the latency bound. The solution (at least for the case where the combining operation is associative and commutative) is to use $k \times k$ loop unrolling, where k different accumulators are used, and then combined at the end (this splits up the critical path!). Similarly, a reassociation transformation can split up the critical path and break the latency bound.

- *Separate Accumulators*: Using k different accumulators inside a loop can aid the pipeline.

Limiting Factors. *Register Spilling.* Occurs when program has a greater degree of parallelism than registers available. Compiler resorts to storing temporary values in memory. *Branch Misprediction Penalties.*

Code profilers (e.g. VALGRIND) collect performance information about a program as it executes. This can help guide optimization.

5 Memory

Tradeoffs. Larger → slower. Faster → expensive. Larger bandwidth → expensive.

Random access memory (RAM) comes in two varieties - static and dynamic (SRAM/DRAM). SRAM is faster and more expensive than DRAM. DRAM is typically used for main memory.

5.1 Volatile

D Flip-Flop (DFF). Fast but expensive, e.g. registers. 27 transistor/bit. Least dense.

SRAM is implemented with a 6-transistor circuit (2 inverters). SRAM will retain its value as long as power is applied and is insensitive to electrical noise.

DRAM stores every bit in a capacitor. Thus DRAM can be made much more dense, but it is sensitive to power fluctuations. In fact, the capacitor will leak every 10-100 ms, so it must be periodically refreshed.

DRAM chips are partitioned into d *supercells* each consisting of w *cells* (bits). Information flows in and out of a chip via *pins*, which carry 1-bit signals. Supercells are addressed in a 2D-array format: (i, j). Every DRAM chip is connected to a *memory controller*, which can transfer w bits to the chip. To read (i, j), the controller sends row address i (RAS), then column address j (CAS) along the same pin. The row address selects row i of the supercell and moves it into a internal row buffer. Then, the column address returns the jth entry of the buffer. Note: Reading the row drained the capacitors, the buffer is used to reset the row. The advantage of the 2D-array is that few pins are needed. However, it requires that addresses be sent individually, slowing down accesses.

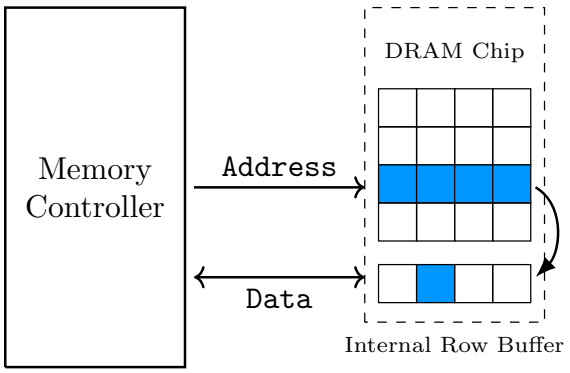
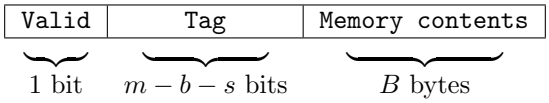


Figure 4: One of E cache lines in a set.



DRAM chips are clustered into *memory modules*. To retrieve a 64-bit word, a memory address A, is converted into a supercell address (i, j), which is passed onto each DRAM. Each of 8 DRAMs then returns 8-bits.

5.2 Non-volatile

- Disk: Stores firmware.
- Flash: Used on phones.
- Tape: Longest lasting.

5.3 Memory Hierarchy & Locality

Spatial Locality. Memory locations nearby a recently referenced memory location are likely to be referenced in the near future. *Temporal locality.* A recently references memory location is likely to be referenced again in the near future. We'll see that using caches which exploit spatial and temporal locality can vastly improve program performance. A *cache* is a small, fast storage device that acts as a staging area for data stored in larger, slower devices. The idea of a memory heirarchy is that at each level k, the faster and smaller storage device at level k serves as a cache for the storage at level $k+1$. Data is transferred between levels in *blocks* of fixed size. When a program needs a data object d, from level $k+1$, it first looks at level k, and if d is cached at level k, then we have what is called a *cache hit*. Otherwise, we have a *cache miss*, and the block containing d at level $k+1$, overwrites some block in level k.

Type	Storage	Where	Latency (cycles)	Managed By
Registers	8-byte words	On-chip	0	Compiler
TLB	Address translations	On-chip	0	Hardware
L1 - L3 Caches	64-byte blocks	On-chip	4, 10, 50	Hardware
VM	4-KB pages	Main memory	200	Hardware/OS
Buffer Cache	Parts of files	Main memory	200	OS
Disk Cache	Disk sectors		100K	

5.4 Caches

Consider a computer system with $M = 2^m$ unique addresses. A cache on such a machine is organized into an array of $S = 2^s$ *cache sets*, each consisting of E *cache lines*. Each line consists of

- a data *block* of $B = 2^b$ bytes,
- a *valid bit*, and
- $t = m - (b + s)$ *tag bits* that uniquely identify the block stored in the cache line.

A given memory address is partitioned, such that the lower order b bits form a *block offset*, indexing into the memory contents of a cache line, the next s bits are used to index into one of the cache sets, and the remaining t bits are the unique tag. A cache with $E = 1$ lines per set is *direct-mapped*. A *set-associative cache* permits $1 < E \leq C/B$. Note the size of a cache is $C = E \cdot B \cdot S$.

5.4.1 Reading from Caches

A word w is contained in the cache if and only if for some line in the set identified by the s set bits in w's address, the valid bit is set and the line's tag matches that of w. Then, we can index into the memory contents of the line using the b lower order bits of w's address (interpreted as an unsigned integer). If there is a cache miss, then we need to retrieve the block containing w from the next level in memory and store it in the cache.

In a set associative cache, the line replacement policy, will replace any empty lines first, and then apply a *least recently used* (LRU) policy, which replaces the line in the cache that was least recently accessed.

Note: There exist software managed caches called scratch-pad memory, but they are difficult to manage and have low portability.

5.4.2 Writing to Caches

We also need a policy about what to do when we write to a memory address in the cache. A *write-through* policy immediately write to the lower level in memory (this increase bus traffic). A *write-back* policy only writes to the lower level when the modified line is evicted (this requires the hardware to manage a *dirty bit*, indicating the line has been modified).

To deal with write misses, a *write-allocate* policy loads the corresponding block from below and updates. The alternative *no-write-allocate* bypasses the cache and writes directly to memory.

5.4.3 Unified & Separate Caches

The question is whether or not to separate instruction caches from data caches. Separate caches lower the risk of conflict misses and allow data and instructions to be read in parallel. The independent caches can also be fine tuned to the different access patterns of data and instructions. Modern processors typically use independent higher level caches and unified lower level caches.

Large Cache Size	Large Block Size	High Associativity
• Better hit rate • Higher access time	• Better hit rate with good spatial locality • Worse hit rate with more temporal locality • Larger miss penalty	• Decreases cache thrashing due to conflict misses • Harder to implement • Larger hit time • Larger miss penalty

Table 1: Caching Tradeoffs.

5.5 Cache Friendly Code

Just think about exploiting temporal and spatial locality, e.g. tiling or loop permutations. Use a data object as many times as possible once it is pulled from memory.

6 Exceptional Control Flow

- *Asynchronous Exceptions.* Also called *interrupts.* Caused by events external to the processor, e.g. timer interrupt, I/O interrupt (Ctrl-C). Indicated by processor's interrupt pin. Handler returns to next instruction.
- *Synchronous Exceptions.*

- *Traps.* Always intentional, e.g. system calls or breakpoints. Return to following instruction.
- *Faults.* Unintentional but usually recoverable, e.g. page faults or segmentation faults. Will either return to current instruction and re-execute or will abort the program.
- *Aborts.* Always unintentional and unrecoverable, e.g. illegal instruction or parity error. Causes program to abort.

Both exceptions and interrupts transfer control to the OS kernel in response to an event. Exception handling code in the kernel deals with the problem. For interrupts, the handler is usually called once the current instructions in the pipeline have been processed. For exceptions, handling is usually immediate. Some interrupts may be *masked* (ignored) by the processor.

6.1 Buses

Allow different components of a computer to share data. Arbitration of the bus is either central, in which case a single hardware unit determines priority, or distributed, in which case every device decides whether it is allowed access to the bus.

Data can be transferred synchronously, where are transfers take a fixed amount of time, or asynchronously via a handshaking protocol.

6.2 Processes

A process is an instance of a program in execution. Each program in the system runs in the context of some process. The context consists of the state that the program needs to run correctly; it consists of the program code and data, its stack, register contents, PC, environmental variables, etc. Each time a user runs a program in the shell, the shell makes a new process and runs the executable in the context of this process. A process provides each program with the illusion it has exclusive use of the processor.

A logic flow whose execution overlaps in time with another is a *concurrent flow* (flow Y starts after X, but before X ends). If two flows are concurrent on different computer we say they are *parallel*. Each process is given its own private address space in memory.

Processors provide a *mode bit*. When on, this bit indicates the current process is in *kernel mode*, so the process can execute any instruction or access any memory location. Otherwise, the process is in *user mode*, where access is restricted. Only exceptions can switch a process from user to kernel mode.

If the kernel schedules a new process to run, it preempts the current process and transfers control to a previously preempted one via a *context switch* that (1) saves the current context, (2) restores the saved context of the other process, and (3) passes over control.

Aside. Unix system-level functions typically return -1 when encountering an error and store error information in a global int variable `errno`. For example, if `foo` is some system function,

```
if((pid = foo()) < 0){
   fprintf(stderr, "Error: %s\n", strerror(errno));
   exit(0);
}
```

provides error checking.

6.3 Unix System Calls

A terminated process (*zombie*) is kept until it is *reaped* by its parent. `init` process is created by the kernel on start-up, never terminates, and is the ancestor of every process. If a parent terminates without reaping its child, `init` will reap for them.

See Appendix A for system call descriptions.

6.4 Signals

A *signal* is a message to a process that an event that occurred in the system. A process can either ignore a signal, terminate, or *catch* the signal by executing a signal handler. Every process belongs to some *process group* (`getpgrp(void)` returns current processes group ID). By default a child process belongs to the same group as its parent. `setpgid(pid, pgid)` changes the process group of `pid` to `pgid`.

- Ctrl-C sends SIGINT signal, terminating every process in the foreground process group. Ctrl-Z sends SIGTSTP signal suspending every process in the foreground.
- See *p. 764* for signal blocking information.
- *Safe signal handling.* Keep handlers short. Only call async-signal-safe functions. Save and restore `errno` in handler. Protect accesses to global data by blocking all signals beforehand, then unblocking afterwards.

6.5 Nonlocal Jumps

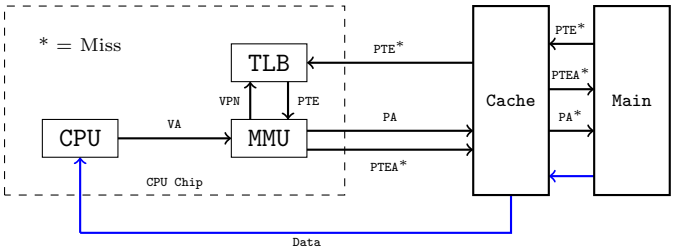
- `int setjmp(jmp_buf env)` Must be called before `longjmp`. Can transfer to arbitrary location. Called once, initially returns 0, but may return multiple times. Calling environment stored in the `env` buffer.
- `void longjmp(jmp_buf env, int ret)` Restores calling environment to `env` and triggers return to most recent `setjmp`. The return of `setjmp` is `ret`.

7 Virtual Memory

The *memory management unit* (MMU) translates virtual addresses to physical addresses. Modern systems typically support $n = 32$ or 64 bit virtual address spaces of size $N = 2^n$ bytes.

VM is partitioned into fixed block called *virtual pages*, with size 2^p. Each page is either *unallocated* (has no associated data), *cached* (allocated memory currently cached in physical memory), or *uncached*. Most SRAM caches use physical addressing. DRAM cache misses are very expensive (disk reads). So DRAM caches tend to be fully associative and use write-back and virtual addressing. A *page table* (maintained by the OS) stores the mappings from virtual to physical addresses. The *page table entries* (PTEs) store valid bits and an n-bit physical address. A null address indicates the virtual page has yet to be allocated. The valid bit indicates whether the page is stored on DRAM or disk.

A DRAM cache miss is called a *page fault*. Page faults trigger an exception. The kernel handles this exception by selecting a victim page in DRAM, which is copied to the disk if it has been modified, then *swapping in* the target page from the disk and updating the PTE. The strategy of waiting until a miss occurs before swapping is called *demand paging*. Every process is provided a separate page table/virtual address space. Page table also maintain read/write bits for memory protection.



The *page table register* (PTBR) points to the current page table. The n-bit VA has two components: a p-bit *virtual page offset* (VPO) and an $(n-p)$-bit *virtual page number* (VPN). The VPN corresponds to a PTE. The physical address is the concatenation of the *physical page number* (PPN) from the PTE and the VPO. The VPO is identical to the *physical page offset*. A *translation lookaside buffer* is a virtually addressed cache of PTEs inside the MMU.

Maintaining a single page table can be memory expensive. A k-level page table heirarchy, partitions the VA into k VPNs and a VPO. Each VPN indexes into a page table at level $1 \leq i \leq k$. Every PTE at level $1 \leq j \leq k-1$ points to the base of some page table at level $j+1$. Each PTE in the kth page table contains either a PPN or an address in the disk. Then, if the PTE in level j is NULL, the corresponding level $j+1$ page table does not have to exist.

7.1 Dynamic Memory Allocation

A *dynamic memory allocator* manages the area of a processes VM called the *heap*. The variable `brk` points to top of heap in each process. DMA is important for when the size of a certain data structure is not known until runtime.

- *Explicit allocators*. Require a program to explicitly free any allocated blocks.
- *Implicit allocators*. Aka garbage collectors, detect when an allocated block is no longer used and free it.

Constraints. *Handle arbitrary request sequences*. The allocator cannot assume anything about the ordering of allocation and free requests (other then that free requests are to previously allocated blocks). *Make immediate responses to requests*. In particular, the allocators cannot reorder requests to improve performance. *Use only the heap*. Data structures used by the allocator must themselves be on the heap. *Alignment*. Must align blocks so that they can hold any data object. *Cannot modify allocated blocks*. Compaction is not allowed.

Goals. *Maximize throughput* (allocate/free operations per second). *Maximize memory utilization*. A request for p bytes, has a *payload* of p bytes. Given a sequence of requests $R_0, \ldots, R_{n-1}$, denote the aggregate payload P_k. Let H_k denote the current size of the heap. Then peak utilization over the first $k+1$ requests is $U_k = \frac{\max_{i \leq k} P_i}{H_k}$.

Internal Fragmentation. Occurs when allocated block is larger than payload. *External Fragmentation*. Occurs when there's enough aggregate free memory to satisfy a request, but no *single* free block is large enough.

- *Implicit free lists*. Data structure to identify free blocks. Each block consists of a header, the payload, and possibly padding. The header encodes the block size and whether it is allocated or freed. Thus free blocks are linked implicitly by the size fields of the headers. This allows the allocator to traverse the set of free blocks by traversing all of the blocks in the heap. We need a special end block (size 0, allocated bit set) to mark the end.

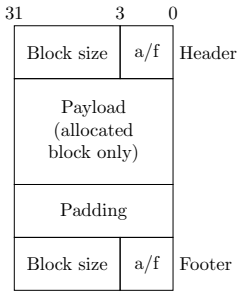


Figure 5: Heap block diagram.

- *Placement. First fit*: allocates first free block from the beginning that fits. *Next fit*: similar to first fit, but starts search where the last one left off. *Best fit*: examine all free blocks, choose one with smallest size that fits.
- *Splitting.* If the selected free block is too big for the allocation request, the allocator may choose to split the block.
- *Coalescing.* False fragmentation is the phenomenon where there are multiple adjacent free blocks. To combat this, allocators implement (immediate or deferred) coalescing. To implement coalescing we use a footer (identical to the header) as in Figure 5. See *p. 852* for more on boundary tags/optimizations.

An *explicit free list* is a doubly linked list whose nodes are stored inside the old payload of freed blocks. One approach is LIFO ordering, which provides constant time freeing and coalescing. Another is address ordering, which increases free time, but provides better memory utilization. In general, explicit free lists increases minimum block size. Multiple free list (segregated lists) with free blocks grouped according to size can reduce allocation time.

7.2 Garbage Collection

Garbage collection is the automatic reclamation of dynamically allocated memory. It is used in Java, JavaScript, Python, Ruby, Lisp, Perl, etc. Consider the view of heap memory as in Figure 6. The root nodes are pointers stored in registers, the stack, etc. to locations on the heap. Nodes are blocks in the heap and edges are pointers. The garbage nodes are marked in red.

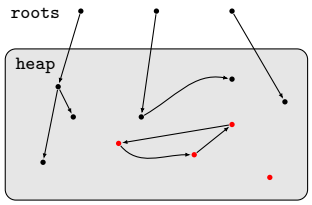


Figure 6: Graphical depiction of the heap.

A *mark & sweep* algorithm is a graph traversal of the heap, starting at the root nodes, which marks all reachable nodes, then scans all the blocks, freeing unreachable nodes. GC is often problematic as the program must occasionally halt to perform GC. So GC injects non-determinism. However, performing incremental GC or concurrent GC can help.

GC is difficult in C since it is difficult to identify whether data stored in memory is a pointer or not. Furthermore, a pointer can point to anywhere in a block making it impossible to find the

header without an extraneous data structure. Other GC algorithms include mark-sweep-compact, generational collectors, and reference counting.

8 Concurrency

8.1 Threads

A *thread* runs in the context of a process. Every thread has its own stack frame and local variables. Unlike processes, threads may share code, global data, and kernel context. Threads are also less expensive to execute than processes. Two threads are concurrent if their logic flows overlap in time; otherwise, they are sequential.

8.2 Semaphores

Semaphores are used to prevent *synchronization errors*. A *mutex* or *binary semaphore* is either 0 or 1, with former indicting that some thread is in the critical region. A P operation decrements a semaphore s if s is nonzero. If s is zero, then P pauses until s is incremented. A V operation increments s upon exiting the critical region. Note semaphores always satisfy $s \geq 0$; in unsafe regions, however, semaphores would be negative.

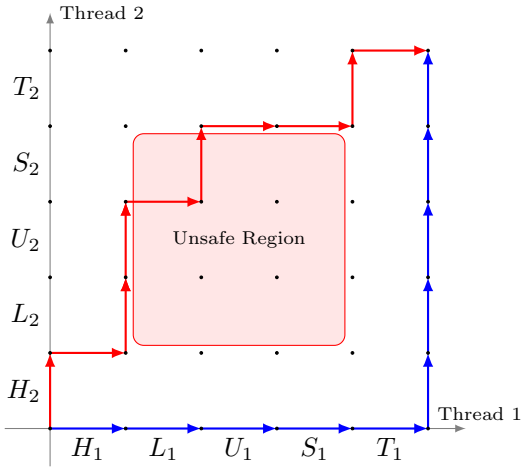


Figure 7: The red path is an unsafe trajectory, whereas the blue path is safe.

A process or thread is in a *deadlock* if it is waiting for a condition that will never be true.

Amdahl's Law. Captures speedup due to parallelism. Let f denote the fraction of a program that is parallelizable and n denote the number of cores. Then the theoretical speedup $S(n)$ is

$$S(n) = \frac{1}{1 - f + \frac{f}{n}}.$$

8.2.1 Single Core Multithreading

Context switching during caches misses gives significant performance enhancement. Switch policy can be fine grained, switching cycle-by-cycle to avoid branch prediction, or coarse grained. Coarse grained policies are either event-based (e.g. on cache miss) or quantum (every fixed number of cycles).

Hyperthreading. If a CPU has multiple functional units (ALUs), then by replicating register files, a single core an process multiple threads simultaneously. Eliminates the need for context switches.

Multicore. Can be symmetric or asymmetric. Asymmetric MCPs have big and little cores, the former consumes more energy but has faster output. Separate cores can have varying cycle frequencies, giving the CPU a larger energy/performance tradeoff space. Symmetric MCPs have identical core types.

Cache Coherence. There are data dependencies between threads in different cores. How do you ensure coherence? Answer: update or invalidate.

For example, in a cache with the invalidate policy, each cache line has three states: modified, valid, and shared. Figure (8) shows the transition diagram.

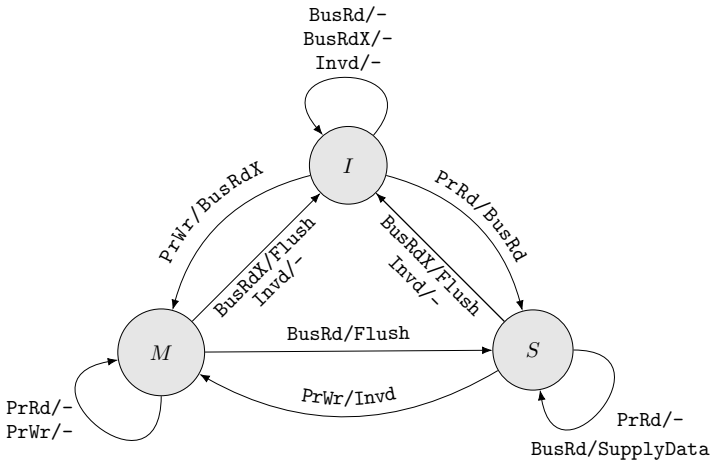


Figure 8: Cache line transition diagram. Syntax: Event/Action

Programmers can also ensure cache coherence if the ISA provides cache flush/invalidate instructions, e.g. the OS manages TLB coherence.

Appendix A.

Process IDs are stored in the `pid_t` data type in C.

- `pid_t getpid(void)` returns the process ID of the calling process.
- `pid_t getppid` returns the PID of parent.
- `void exit(int status)` terminates current process.
- `pid_t fork(void)` allows a parent process to create a new (copied but disjoint) child process, returns 0 from child, PID of child to parent. See *p. 742* for process graph.
- `pid_t waitpid(pid_t pid, int *statusp, int opt)` suspends calling process until a child process terminates, returns child PID. If `pid > 0`, the wait set is the process whose PID is `pid`. If `pid = -1`, the wait set is all of the parent's children. See *p. 744* for more.
- `pid_t wait(int *statusup)` equivalent to `waitpid(-1,&statusup, 0)`.
- `sleep` pauses program for `arg` seconds, should return 0. `pause` pauses program until signal is received, returns -1.
- `execve(const char *f, const char *argv[], const char *envp[])` loads executable file `f` with argument list `argv` and environmental variables `envp`. Doesn't return if call exits normally.
- `int kill(pid_t pid, int sig)` Sends signal type `sig` to process `pid`. See *p. 761.*
- `alarm` Allows a process to send SIGALARM to itself. See *p. 762.*
- The default action associated with a signal can be changed with `signal`.

Appendix B.

Static linkers take as input a collection of relocatable object files and command-line arguments and generate as output a fully linked executable. Linkers have two main tasks:

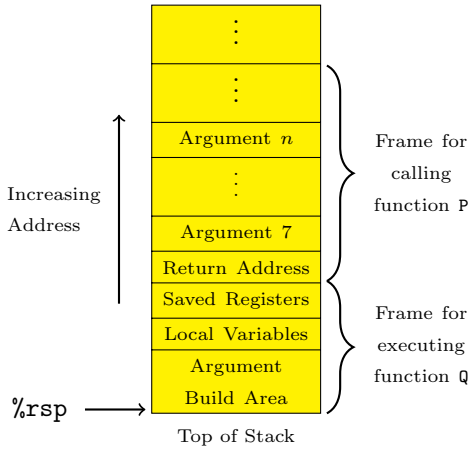
- *Symbol Resolution.* Object files define *symbols* corresponding to functions, global variables, or static variables. Symbol resolution associates each symbol reference with exactly one symbol definition.
- *Relocation.* Each symbol definition is associated with a memory location.

A typical ELF relocatable object file contains the follow sections among others:
`.text` The compiled machine code.
`.rodata` Read-only data, e.g. format strings in `printf` or jump tables.
`.data` Initialized global and static variables.
`.bss` Uninitialized global and static variables or those initialized to 0.
`.symtab` Symbol table.
`.debug` Debugging symbol table, only present with `-g` option.

Appendix C.

- `void* malloc(size_t size)` returns a pointer to a block of at least `size` bytes. `calloc` initializes this memory to zero. `realloc` allows an application to change the size of a previously allocated block.
- `void* sbrk(intptr_t incr)` grows/shrinks heap by adding `incr` to the `brk` pointer.

Threads are managed using the POSIX thread libraries, semaphores are managed using `semaphores.h`.



References

[1] Bryant and O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd ed.

All page references are to [1].