- Program Optimization
    - Improving code at the Assembly level
        - Using registers instead of the stack
        - Limits: callee-save registers: all registers except the **a**, **c**, and **d** registers.
        - Avoiding duplication of effort (as in division).
        - Eliminating useless cases of storing a value from a register to the stack and then immediately copying it back to a register.
    - Larger-scale improvements—at the source code level
        - Hoisting a function call out of a loop
            - But not if the function call has *any* side effects
                - Side effects include file operations, including printing, which affect the real world
                - Side effects include changes to the value of a global variable
        - Simplifying operations involving storage of data to memory or retrieving from memory
            - But not if there is a risk of *memory aliasing*, for instance, if two parameters are addresses to memory locations, which may turn out to point to the *same* location or to overlapping regions
- Memory Hierarchy
    - The structure of the hierarchy, down from fastest and most expensive to slowest and cheapest
    - Structure of a cache
        - How the cache retrieves a word—see the algorithm sent out in an e-mail
        - Be sure to consider caches that are not direct mapped, where the associativity is greater than 1.
            - Remember: associativity is the number of lines per set.
        - Rationale for the structure—why we use the middle bits as set bits, why we use tag bits, why the block bits should be the lowest-order bits.
    - Reasons for using a cache—the rationale of the cache.
- Exceptional Control Flow
    - The sequence of events on an exception
        - Transfer of control from the user code to the kernel code
        - Change of mode from user mode (restricted) to kernel mode (privileged)
        - Kernel transfer of control to an exception handler function according to the exception number
            - How to call a function in a table (array) by index
    - The types of exception
        - Interrupt — asynchronous — triggered by a signal from hardware on the CPU's interrupt pin — recoverable — resumes with the next instruction
        - Trap — synchronous — triggered by an instruction such as **syscall** — recoverable — resumes with the next instruction
        - Fault — synchronous — triggered by hardware, e.g., the memory subsystem, in immediate response to an instruction — possibly recoverable — resumes with the *same* instruction, or else aborts (or triggers a signal that aborts)
        - Abort — synchronous — triggered by hardware in response to an instruction — never recoverable — aborts.
    - System calls and system call wrapper functions — requests for services from the kernel

- Process management
  - What exactly is a process?
  - **fork** and its semantics—its return value—cloning the calling process's address space
    - What exactly happens when a program calls **fork**? Remember that the child process inherits the identical stack, registers, etc., with the IP pointing to the same next instruction. Without conditional code (**if (pid == 0)**), the child and parent processes do *exactly the same thing*, because the code is identical.
  - **waitpid** and its semantics—pausing until the child process has terminated
  - **execve** — causes the IP to jump from the current code (inherited from the parent) to that of a different program. Control *never returns* to the code that issues the **execve** call.
  - **exit**
  - Context switches
    - Why does the kernel ever need to switch contexts?—because there are almost always more running processes than there are available processors to process them.
    - What happens in a context switch (at a high level)?
- Signals
  - What is a signal?
  - What is the first thing the kernel does in order to "send" a signal to a process?
  - How does a process "send" a signal to another process?
  - What is a signal handler function?
  - How does a process "receive" a signal?
    - This just mean that, at the next opportunity, the kernel calls the signal handler for that signal in its signal handler table (array), which is indexed by signal number.
  - How do you install a custom signal handler?
  - Are there any signals for which you cannot install custom signal handlers?
    - Yes: KILL and STOP. This makes sense—otherwise, you'd have the risk of installing signal handlers in such a way that the process can *never* be stopped without your shutting the machine down!
  - What causes **waitpid** to return?
    - The parent process receives a CHLD signal.
- Please make sure you remember what pipelining is and why it might be beneficial in system design. Remember what throughput and latency are.

**creat( ), open( ), close( )** -- managing I/O channels
**read( ), write( )** – handling input and output operations
**lseek( )** – for random access of files
**link( ), unlink( )** – aliasing and removing files
**stat( )** – getting file status
**access( ), chmod( ), chown( )** – for access control

**exec( ), fork( ), wait( ), exit( )** --- for process control
**getuid( )** – for process ownership
**getpid( )** -- for process ID
**signal( ) , kill( ), alarm( )** – for process control
**chdir( )** – for changing working directory

**mmap(), shmget(), mprotect(), mlock()** – manipulate low level memory attributes

**time(), gettimer(), settimer(),settimeofday(), alarm()** – time management functions

**pipe( )** – for creating inter-process communication

caller-save register needs)

-base address of an array

-index of an array (max_count)

-current max count

-ascii code for the data of the array

///IDIV: edi - dividend, esi - divisor

Side effect? When and why? Code motion?

Side effects: Opening a file, printing to the command line, exiting a program, and so much more.

- "Unsafe" considered by the compiler

- "Hoisting": compilers cannot hoist code (moving code = code motion)

Rotation disk == SSD in terms of speed/costs (actually, SSD is faster in real life)

Register > SRAM(Cache) > DRAM(Main Memory) > Local Disk(SSD) > File System/Web Server

Caller/Callee save registers: caller(a, c, d registers), callee(si, di, %rbp, %rsp, b registers)

Write back/Write through)

- Write back: Copy the code immediately (modern computers follow write-back policy)
- Write through: Copy the code only when it needs to be evicted
- "Write back" is likely to incur less overhead when a program needs to write to memory frequently

Capacity of cache = L x B x S

Examples of exceptional control flow)

- Aborting a program because it attempted to divide a number by zero.
- Printing a character to the console. (getchar)
- Responding to a hardware interrupt

Exception cases and examples of each (getchar = trap systemcall)

- Copying a file from disk to main memory
- Retrieving a web page
- Multitasking

Kinds: Interrupt(I/O device, CPU's interrupt pin, Trap(system call), Fault(page fault), Abort(data corruption)

Temporal locality vs. Spatial locality)

- Temp = how likely a datum will be re-used

- Spat = the degree to which the use of one datum predicts the use of data stored near it(use of data clumped together -> greater spatial locality)

Locality vs. modularity ///

first parameter of function as an int --> stored in %edi ///

Pipelining, throughput, latency)

Pipelining vs. Sequential Execution

Pipelining = Increases throughput, but the progress is limited to the "slowest" process; faster processes can't skip the slower ones; the extra time to store data in the pipeline registers increases overall latency/

Throughput = The number of operations completed in a given amount of time

Latency = The average amount of time required for an operation.

Fetch -> Decode -> Execute -> Memory read/write -> Write register -> IP update

Impossible to override the default signal handler: STOP, KILL

System call wrapper functions: waitpid, exit, execve, fork, open, close, installing a signal handler

Context Switch = A context switch is occasioned by an interrupt— • From I/O hardware, when a long-running task is completed • From the system timer, which issues an interrupt at regular intervals.

Process Context = When the kernel interrupts a process to switch to another one, it saves essential state data in a complex data structure called the process context; the context contains information necessary to recreate the state of the interrupted process so that it can resume as if it had never been interrupted!

Process - can be running / stopped / terminated

Exception Handler - when it's done, the execution may continue in the current instruction / next instruction / Abort

fork & child/parent & PID

Non-volatile storage: rotating disk, SSD (usually read only & data saved even without power)

Volatile storage: SRAM, DRAM (read & write & data not preserved when shut off)

Virtual memory uses main memory as a cache for non-volatile storage

Page faults are very expensive because disk access and copying time are so much slower than memory access time.

Main memory is a fully associative cache for the disk—any page on disk may be stored in any page-size region of memory.

• This minimizes the occurrence of empty memory regions and therefore missing pages.

• Temporal locality helps keep performance good: the more a program re-uses the same instructions or data within a page, the less likely it is to need to swap a new page in.

MMU = Memory Management Unit

The page table is created by the kernel and stored, itself, in main memory.

• The MMU holds the base address of the page table in a control register: the Page Table Base Address (PTBA) register.

• Each page table entry (PTE) represents a page of virtual memory—indexed by page number.

The page number is the index into the page table.

Each PTE holds either • The

m − p high-order bits of the physical address, or physical page number — OR

• The address of the start of the page on disk, using whatever scheme the hardware requires — OR

• 0, indicating that the virtual page is not mapped at all.

• In addition, each PTE holds control bits, including • A valid bit, which is set if the contents hold the physical page number • A dirty bit, showing that the page (if present) has been changed by writing.

Physical memory address = physical page number(high-order bits) + original virtual page offset(low-order bits)

If MMU triggers a page fault, the CPU switches to kernel mode and jumps to the kernel's exception handler.

If the PTE entry has 0, the page is unallocated or unmapped—the kernel issues a segmentation fault signal.

• The kernel also checks access permissions on the page—to be discussed later.

• If those tests pass, the kernel finds an empty page of physical memory (unlikely) or a page to replace (likely).

• If the "victim" page is dirty, the kernel first calls I/O routines to copy it back to disk. (This is the write back data storage policy in caches.)

• The kernel uses data structures to keep track of used and available DRAM pages.

• The kernel may use extra control bits to determine the least recently used page or implement some other algorithm for choosing a victim for replacement.

• The kernel calls I/O routines to copy the page from disk to the vacated region of memory.

The kernel replaces the disk address with the DRAM page number in the PTE for that virtual page.

• Since the I/O routines take a long time, the kernel is likely to have paused the current process until both writing back the old page (if necessary) and copying in the new page are complete.

• Once the page is copied, the kernel restarts the process by repeating the faulting instruction.

• This time, the CPU's request for a word leads to a page hit.

practice of delaying the swapping of pages as long as possible is called demand paging.

Allocating a page means mapping a virtual page to a page of data on disk.

• Once the page is allocated, code can refer to a word in it. This would trigger a page fault.

• The page fault would result in copying the page into main memory. • The kernel allocates pages— • When its loader loads a program from disk in order to launch it as a process • When a user program makes a system call through mmap.

Each PTE may have additional information to control access to the page— • Restriction to kernel mode access • Read-only or read-write access • Execution access

• The kernel checks access while servicing a page fault.

• Attempted invalid access to a page results in a general protection violation, which the kernel signals as a segmentation fault.

Each process has its own page table.

• Process A's PTEs are not mapped to private physical pages belonging to Process B.

• Thus Process B's data are hidden from Process A.

• However, read-only data may be shared—both Process A and Process B may have mappings (typically from the same virtual addresses) to kernel and library code

A program calls fork to ask the kernel to create a new process. • The kernel copies the page table of the parent process, but sets all pages as read-only.

• When the child process writes (stores) a datum on a page for the first time, this causes a page fault.

• The page fault handler function in the kernel copies the page to a new region of physical memory and updates the child process's PTE for that page.

• This lazy approach to copying data is called copy on write. It copies data only when needed.

On the execve call, the loader code in the kernel replaces the PTE contents with disk addresses from the program file.

• Pages will now be copied lazily into main memory after page faults.

The page table must itself be stored in main memory.

• But the running process needs page translations frequently and fast—loading translations from main memory could slow execution down.

Instead of storing a single page table in memory, we store a table of tables.

• Each PTE in the top-level table represents a chunk of pages and has the physical address of a page table for that chunk.

• If none of the pages in the chunk are allocated, the page table does not exist. We simply mark the top-level PTE for the chunk as unallocated.

• This works especially well because there are large holes in the virtual address space by operating system convention—so there are large ranges of addresses that are always unallocated when a process starts.

• Most of these pages never get allocated at all

A 64-bit system with 4KB pages could have page tables with 512 PTEs of 8 bytes each. This would preserve the rule of a page table taking up one page.

• The index for a page table at each level would require 9 bits. The MMU breaks the address up into 9-bit segments for the successive page table indices.

• For a full 264 -byte virtual address space, page numbers would require 64 − 12 = 52 bits, requiring 6 levels of page table.

• Modern 64-bit Intel machines actually use 48-bit or 57-bit addresses, and 4 or 5 levels in their page table hierarchies.

To speed translation, the MMU has a small cache of PTEs, called the Translation Lookaside Buffer (TLB).

• This is structured just like any other cache—like the data and instruction caches.

• A block may consist of several contiguous PTEs.

• The cache returns a single PTE instead of a word.

• The MMU looks for the PTE first in the TLB—where the virtual page number is split into tag, set, and block bits.

• Again, this technique favors good temporal and spatial locality—at the page level

Dynamic memory is memory allocated when needed and in the quantity needed (or close to it), by program request.

• Dynamic memory is also de-allocated when it is no longer needed.

• The stack is not dynamic memory because it is a range of addresses that is preallocated and whose size cannot be changed during the course of the program.

• A stack frame is not dynamic memory because it is automatically managed—a program cannot request (or decline) a stack frame.

• Static memory is not dynamic because it remains in place from program start to termination.

The region of memory used for dynamic allocation and de-allocation is called the heap.

• It is also called the free store.

Purpose of using dynamic memory: How much storage a program needs may not be known at development time.

• E.g., the quantity of input may itself be an input—the user enters the number n of numbers and then enters n numbers.

In addition, stack objects are de-allocated automatically when functions return. Dynamic memory allows objects to persist in memory until they are no longer needed.

• You can allocate whole pages of memory with mmap, which persist until you call munmap, but allocation by whole page units could be wasteful

Dynamic memory is managed by library code, such as the malloc package in C, which is declared in stdlib.h.

• Such library code is called an allocator. • Allocators may be explicit or implicit.

• An explicit allocator requires that the program explicitly de-allocate dynamic memory when it is no longer needed — e.g., by the free function call in C or the delete keyword in C++.

• An implicit allocator uses routines that run automatically at certain points in the program to de-allocate memory that they can determine is no longer needed.

The malloc package consists primarily of four functions:

• void * malloc(size_t size); If successful, returns the address at the start of at least size bytes of available memory storage. Returns NULL if unsuccessful. The return type is void *, an untyped pointer, which should then be cast to the appropriate type.

• void * calloc(size_t count, size_t size); If successful, returns the start address of count times size bytes of available memory initialized to zero — and NULL if unsuccessful.

• void * realloc(void * pointer, size_t size); Tries to change the size of the memory block beginning at pointer. If not enough memory is available, it finds a different block of the requested size, copies the data over, and frees the original block. Returns NULL on failure.

• free(void * pointer); CSCI 2271 – Computer Systems – Boston College

De-allocates the block of memory starting at pointer so that it can be used by some other allocation request.

When the loader code in the kernel maps out the process image before the process starts, it sets aside an unmapped region immediately above bss (uninitialized global variables) for the heap.

• It sets the top (the end) of the heap, called brk (“break”), at the same point as the start.

• The allocator asks the kernel to move brk up by page-sized increments with the sbrk system call wrapper function.

• It also calls mmap for those pages as demand zero pages.

• It then manages the available heap by allocating blocks from it.

Allocator Requirements)

• Handle arbitrary requests to allocate and free in any order.

• Respond immediately rather than delaying requests and then handling them all at once. The allocator may not rearrange requests for efficiency.

• Keep all metadata on the heap itself. This makes the allocator portable and scalable.

• Align the block — generally to an 8-byte boundary. This makes memory access faster and makes it easier to compute available memory.

• Do not modify allocated blocks.

Goals of Allocator)

• Maximize throughput. Functions should return quickly with the requested memory.

• Maximize utilization. Memory should not be wasted. Allocate a block greater than or equal to the size of the request, but as close as possible to that size.

• These two are in tension. For example—

• If the allocator returns the first block of memory it finds that is big enough, it may be too big, leading to poor utilization.

• If it looks through all available blocks to find the best fit, it takes longer, so utilization improves but throughput suffers.

Main cause of poor heap utilization: Fragmentation

Fragmentation is the breaking up of available free memory so that it cannot be used.

• Internal fragmentation is when too big a block is allocated for a given object (the payload). The excess storage space is wasted. This may happen because of alignment or because the allocation algorithm chooses a block that is big enough rather than one that is also no bigger than necessary.

• External fragmentation is when there is enough aggregate free memory available for a request, but it is not contiguous and therefore cannot be used.

Typically, the allocator breaks the heap up into blocks, with metadata stored along with the blocks. • A simple arrangement has the block size in a header, along with a bit telling whether it has been allocated. • Because of 8-byte alignment, the last 3 bits of the size are always 0, so the last bit can be used to mark allocation.

• The allocation routine can use the size to find the next block, traversing the sequence of blocks as if it were a linked list to look for a suitable free block.

In order to try to improve utilization, the allocator may split a free block into an

aligned block at or just above the required size and a free remainder to be used later. • The allocator starts with one big, unbroken block. • As requests come in, it carves out blocks on demand.

• As blocks get freed, it starts to have free blocks mixed in among allocated ones.

• It may break up any such free block in order to fulfill a request.

• This may reduce internal fragmentation by fitting the block to the request, but increase external fragmentation by breaking blocks up.

To decrease external fragmentation: coalescence)

As blocks get freed, the allocator may join two or more free blocks if they are contiguous.

• In a variant design, each block has a footer, with a copy of the header information, to make it easy for the allocator to check a previous block and coalesce it with a block currently being freed.

Explicit Free lists)

Popular allocators today usually use double-linked lists, in which each free block has a pointer to the next free block.

• Since those blocks are free, there is room to store addresses as well as sizes. • Allocated blocks have size headers but no pointers.

• The list may be kept in last-in, first-out (LIFO) order. When a block is freed, it is put in the front of the free list.

• With coalescence, this can make new allocations, as well as de-allocations, run in constant time.

The malloc package is an explicit allocator, meaning that it requires de-allocation of allocated blocks by means of calls to free.

• Languages such as Java and Python have implicit allocation, where they periodically de-allocate unused dynamic memory.

• This periodic process is called garbage collection.

• Allocated memory should be de-allocated when the object occupying the memory can no longer be reached by any code.

• In high-level languages, hidden metadata keep track of objects and when they become unreachable— e.g., after a class destructor is called.

• C avoids the overhead of such runtime management—but this means that garbage collectors must find unused allocated blocks, and must err on the side of caution if they cannot prove that an object cannot be reached

Garbage Collecting Algorithm)

The first and best-known garbage collection algorithm is mark and sweep. • First mark every block that can be reached from any pointer in the program.

• This includes, not only the object to which the pointer points, but any object to which any pointer within that object may point, etc.

• Then de-allocate all unmarked blocks.

Mark phase)

Go through every stack frame by pointer-sized unit. • Check whether the contents are the address of an allocated block • Check whether the contents could be an address—within range • Check whether the corresponding block is marked as allocated.

• If so, mark the block as reachable. • For each block in the heap, if it is marked reachable, • Go through each pointer-sized unit to check for an address of an allocated block and • Mark as reachable accordingly.

Cautiousness!!!

The compiled code does not preserve any type information. • An address is just an integer.

• A word-sized datum may happen to coincide with a plausible address, and the block containing that address may be allocated—and therefore seemingly reachable!—even if, in fact, the block is unreachable.

• This would be a false positive—the block is unreachable, but a mere integer happened to look like an address pointing to it.

The page tables needed for the virtual memory translation scheme must, themselves, be stored in memory. (They may be impossibly big. • Access to the translation may be unacceptably slow.)

To solve these problems, we use multi-level page tables to save space and make tables manageable.

• The MMU uses a cache, the Translation Lookaside Buffer (TLB), to speed up frequent lookups.

Dynamic memory enables programs to allocate and de-allocate memory storage space whenever, and in whatever quantities, it needs.

• Explicit allocators such as the malloc package require that the program free memory it no longer uses.

• Implicit allocators use garbage collection to recycle memory no longer needed.

• The allocator acquires memory for the heap from the kernel with sbrk and mmap.

• The allocator carves the heap up into blocks and then splits and coalesces blocks as needed.

The allocator aims to balance good throughput against good utilization. • Blocks are arranged to form a free list, which may be implicit, using only size headers, or explicit, using pointers to predecessors and successors.

• Mark and sweep is an algorithm to find unreachable blocks for garbage collection.

• In the absence of runtime type metadata, mark and sweep is cautious and safe.

An execution flow is a sequence of operations to be executed. These may include code instructions as well as hardware actions.

• Execution flows are concurrent if they overlap in time.

• Concurrency is the condition of having concurrent operations.

Another word for concurrency is parallelism (or parallel processing).

Patterns of flows: simultaneous & interleaved

Simultaneous examples)

Processes on a multi-core machine—each process runs concurrently on a different CPU.

• Retrieving a page from disk for one process while the kernel runs another process.

• Pipelining—different stages of the processor run different instructions at the same time.

Interleaved examples)
• Processes on a single-CPU machine—the kernel scheduler rotates execution.

• Hyperthreading—two processes share the same hardware and take turns—the CPU alternates between them.

Benefits of concurrency)
Having processes or other execution sequences run concurrently saves time.

• It improves the utilization of hardware resources—though it increases power consumption per unit time.

• Because it allows us to split tasks and run them at the same time, it can make computers more responsive and interactive—e.g., if one process computes data while another process handles I/O traffic.

• Because it enables the computer to do more work in the same allotted time, it allows clock speeds to be scaled back while still improving performance.

• This reduces heat production and power consumption per task

Kinds of concurrency)

• Processes The main process creates one or more child processes, which inherit their own private copies of their parent's state and address space. (Fork call)

• I/O Multiplexing The single process switches control flow to do work while awaiting arrival of data from an I/O channel such as a file on disk or a network connection. All code shares a single address space.

• Threads The process creates one or more threads of execution, each of which has its own private stack and register file—but all threads share global and other static variables and the heap. (partially shared address spaces)

the threading module in Python actually supports I/O multiplexing and not thread-based concurrency.

• The Python threading module is useful for network-bound programs, where tasks need to await the completion of network traffic. The module's routines enable the program to get other work done while waiting.

• Python has no support for actual execution threads. It does have support for multiprocessing through the multiprocessing module.

A widely-supported, standard way to create and join threads of execution is the Pthread API.

• The name Pthread stands for POSIX Thread.

• POSIX — "Portable Operating System" + "ix" (as in Unix) — is a 1988 IEEE standard with which many versions of Unix, Linux, and other OSs comply.

• Create a thread with pthread_create, which takes a function as an argument. The new thread executes that function and then terminates.

• Join a thread to its parent thread with pthread_join — or terminate it early with pthread_kill.

For both concurrent processes and concurrent threads, the kernel must schedule processing time on an available processor.

• If two processes or threads need to be run and there are two available processors at the same time, the processes or threads run simultaneously.

• If, however, there is only one processor available, the processes or threads must take turns—so they are concurrent but interleaved.

• For programmers, the kernel provides an abstraction of concurrent processes or threads as if they were all running at the same time.

Sharing data in static and dynamic storage makes it easy for threads to pass data to each other. • However, it also makes it easy to forget to synchronize accesses to data. • The result may be a race condition. • In a race condition, the order in which threads have access to a shared datum determines the output or behavior of the program—but the program does not determine the order of access.

• The result is mysteriously determined by outside factors such as hardware timing—and may be different on each run of the program.

• A program may give correct results 99 times and then a wrong answer the 100th time—with no warning.

CSCI 2271 – Computer Systems – Boston College

• We call this uncontrolled behavior nondeterministic behavior.

Mutual Exclusion Lock(Mutex) : forces only one thread to act at a time. • Therefore, this technique is sometimes called serialization—it turns parallel code into serial code.

Speedup (S) is the ratio of the old to the new running time: S = T(old) / T(new)

T(1) = serial running time

## Amdahl's Law

### The General Rule

- Let $f$ denote the *fraction* of the program that has been improved.
- Let $k$ be the speedup of the improved fraction.
- Then $T_{\text{new}} = (1-f)T_{\text{old}} + f\frac{T_{\text{old}}}{k} = \left((1-f) + f/k\right)T_{\text{old}}$
- And $S = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{1}{1-f+f/k}$.

In an ideal scenario, the processes divide the work up perfectly evenly.

• There is no part of the program that cannot be so distributed. • • This is called linear speedup.

S(p) = p

• We can approached this ideal in certain "embarrassingly parallel" algorithms —including parallel matrix multiplication.

**Example 2:** 10% of a program's execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

$$S(\infty) \leq \lim_{p\to\infty} \frac{1}{0.1 + \frac{(0.9)}{p}} = 10$$

As the number of processes increases, speedup increases, but with eventually diminishing returns.

What limits concurrency, and what must remain serial:

Communication — the transfer of data to and from storage.

• Management — contexts and schedules, cloning data, etc.

• Synchronization — the serialized access to shared data.

A program that calls many different functions has poorer spatial locality than one that calls fewer functions, but probably has better modularity and readability.

The allocator provides greater throughput at the cost of poorer utilization if the allocator always returned the first free block it encountered whose size is aligned and greater than or equal to the requested size.

Why are concurrent threads considered "lightweight" compared to concurrent processes? That is, why is thread creation thought to incur less overhead than process creation?

Thread creation does not require any substantial copying of data.