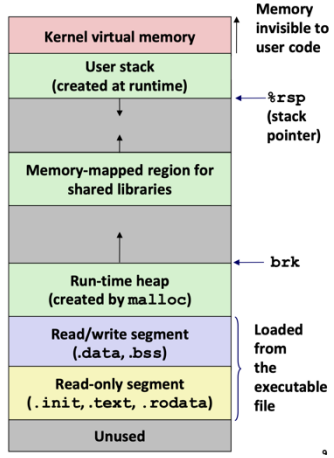
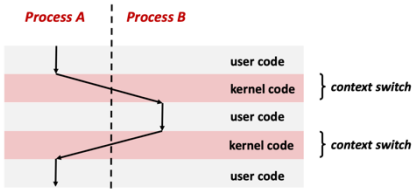
```
short arr[] = {0x1234, 0x8326, 0x9742, 0x4200, 0x1521, 0x3531};
printf("2: %lx\n", *((long*)&arr[2]));           // 3531152142009742
printf("3: %x\n", (unsigned char)*((char*)&arr[3]));             // 0
printf("4: %x\n", (unsigned char) *(((char*)arr)+ 3));           // 83
printf("5: %x\n", ((int*)arr)[1]);                          // 42009742
```

- When **casting** between signed and unsigned types of the same width, the bit representation is kept and reinterpreted. When MSB is 0, the values are identical. When MSB is 1, values have a net difference of 2^w: in a signed number, set sign bit adds -2^w-1; in an unsigned number, adds 2^w-1
- GOTO: equivalent of "condition ? true : false; in C"
  - Both values get computed, may be unsafe/side effects
  - GCC uses them only when known to be safe
  - Why? **Branches** are very disruptive to instruction flow through pipelines, Conditional moves do not require control transfer
- Caller Saved ("Call-Clobbered"): Caller saves temp. values in its frame before call
- Callee Saved ("Call-Preserved"): Callee saves temporary values in its frame before using, Callee restores them before returning to caller
- System-level protections: randomized stack offsets (stack repositioned each time program executes, difficult to predict beginning of inserted code), non-executable memory, stack canaries (check if any region beyond buffer on stack is corrupted)
  - Return-Oriented Programming uses gadgets, does not overcome canaries
- **Padding** is important because memory is accessed by chunks, in the forms of cache lines and VM pages. It would take more cycles to read information if it was not aligned, because data being accessed may straddle two cache lines or pages. In addition, the size of a struct has to be a multiple of its alignment because padding cannot be inserted between the elements of an array (this is a rule of C). Therefore, if a struct's size were not a multiple of its alignment, the second element of an array of those structs would be misaligned
- **TLB**: (ex: 2-level page table) a single translation takes one access to the page directory and another access to the page table entry (L1) to find out the physical address. Moreover, these accesses are from the main memory which is not located so close to the CPU. TLB is located on the CPU, it has really low access time. If the virtual-to-physical mapping is present on the TLB itself, we do not have to go all the way to the main memory twice to translate the virtual address.
- A *k*-level page table requires *k* memory loads in order to determine the physical address. There is no spatial locality to these loads
- **Cold** (compulsory) miss: first reference to the block; **Capacity** miss: additional misses from finite-sized cache, when the set of active cache blocks (working set) is larger than the cache; **Conflict** miss: additional misses due to placement policy; level *k* cache large enough, but multiple data objects all map to same level *k* block
- **Principle of Locality**: Programs tend to use data and instructions with addresses near (spatial) or equal (temporal locality) to those they have used recently
- Caches take advantage of **temporal locality** by storing recently used data, and **spatial locality** by copying data in block-sized transfer units
- Locality reduces working set sizes, most effective when working set fits in cache
- **Memory mountain**: measured read throughput (number of bytes read from memory per second) as a function of spatial and temporal locality
- Write-hit policy: **write-back** (defer write to memory until replacement of line, dirty bit set if data written to), **write-through** (write immediately to memory)
- Write-miss: **write-allocate** (load into cache, update line in cache, good if more writes to the location will follow; if eviction and dirty bit set, write original block to memory), **no-write-allocate** (writes straight to memory, no loading into cache)
- **Benefits of VM**: uses main memory efficiently (use DRAM as a cache for parts of a VA space), simplifies memory management (each process get same uniform linear address space) isolates address spaces (one process can't interfere with another's memory, user program cannot access privileged kernel information and code)
- **Linking**: each program has similar virtual address space; code, data, heap always start at same addresses
- **Loading**: **execve** allocates virtual pages for .text and .data sections and creates PTEs marked as invalid. The **.text** and **.data** sections are copied, page by page, on demand by the virtual memory system



- **execve**: if FD_CLOEXEC flag set with fcntl() or open(), file is closed call, entry is removed from file descriptor table and refcnt decremented in open file table
- If malloc return a NULL pointer, possible ECFs:
  - First array access: NULL is a 0x0 pointer, result in a segmentation fault, signal would be SIGSEGV
  - Alternatively, fault exception occurs due to an invalid memory reference: accessing a NULL pointer + trying to write into it
  - Page fault exception occurs: malloc returns valid pointer and program tried to access memory not yet mapped into physical memory
- **Optimize for branch predictions** (machine-dependent): reduce number of branches (transform loops, unroll loops, use conditional moves*), make branches predictable (sort data, avoid indirect branches like function ptrs/virtual methods)
  - **Unrolling**: amortize cost of loop condition by duplicating body, prepares code for vectorization, can hurt performance by increasing code size
  - **Scheduling**: rearrange instructions to make it easier for the CPU to keep all functional units busy, e.g., move all the loads to the top of an unrolled loop
- **Internal fragmentation**: payload smaller than block due to *previous* requests (overhead or padding); **external fragmentation**: enough aggregate heap memory, but no single free block is large enough (pattern of *future* requests)
- Processes are managed by **kernel**: a shared chunk of memory-resident OS code

- Single processor executes multiple processes concurrently: process executions *interleaved* (multitasking), address spaces managed by VM system, register values for nonexecuting processes saved in memory
- **Control flow** passes between processes via a **context switch**: load saved registers, switch address space by updating Page Table Base Register (address of L1 page table)
- **Multicore processes**: multiple CPUs on chip
  - Share main memory (and some caches), each can execute a separate process
  - Scheduling of processes onto cores done by kernel



- **sbrk**: allocate RAM, grow/shrink heap; change location of *program break*, which defines the end of the process's data segment (memory allocated to process)
- **pid_t wait(int* status)** – suspends current process until any child terminates, returns PID of child, records exit status in status
- **pid_t waitpid(pid_t pid, int *status, int options)** – if pid = -1, any child process; if pid > 0, single child process with given pid; if pid = 0, child with pgid equal to the calling process; if pid < -1, any child process with pgid = -pid
- **int setpgid(pid_t pid, pid_t pgid)** – if pid = 0, pid of calling process is used; if pgid = 0, then pgid of process specified by pid is set to the pid
- Kernel sends **signal** to destination process, which either **ignore**, **terminate**, or **catch** the signal by executing handler (a user-level function)

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-C
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALARM	Terminate	Timer signal
17	SIGCHILD	Ignore	Child stopped or terminated

- **Pending**: signal sent but not yet received, at most one pending signal of each type
- **Blocked** signals can be sent, but will not be received until the signal is unblocked; some signals cannot be blocked (SIGKILL, SIGSTOP) or can only be blocked when sent by other processes (SIGSEGV, SIGILL, etc.)
- **int sigaddset(sigset_t *set, int signo)** – adds signal to signal set
- **int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)** – if SETMASK, set of blocked signals set to argument, if BLOCK, union of arg. and current set
- **int sigsuspend(const sigset_t *mask)** – temporarily replaces mask and suspend thread until delivery of signal that invokes signal handler/terminates a process
- Function can be called in signal handler if **async-signal-safe**: if either reentrant (e.g., all variables stored on stack frame) or non-interruptible by signals
- **int sem_wait(sem_t *s);** /* P(s), "lock", waits for s > 0, then s-- */
- **int sem_post(sem_t *s);** /* V(s), "unlock", s++, resume one thread waiting inside P */
- **int sem_init(sem_t *s, int pshared, unsigned int val)** – if *pshared* = 0, then s is shared between threads of a process; else, s is shared between processes

- **Deadlock**: programs wait for event which cannot happen, no thread can advance
- **Livelock**: threads advance but make no progress towards their goal
- **Starvation**: one process runs continuously and thus other make no process
- **Clients** use URL prefix: server protocol, where it is, and what port its listening to
- **Servers** use suffix to determine if request is for static or dynamic content and find file on file system (initial "/" denotes home directory)
- HTTP request is a request line, followed by zero or more request headers
- <method> <uri> <version> - URI is URL for proxies, URL suffix for servers
- HTTP response is a **response line** followed by zero or more **response headers**, blank line "\r\n", possibly followed by **content**
- <version> <status code> <status msg> - numeric status code corresponds to text