Compile to Assembly gcc -std=c17 -Wall -pedantic-errors -g -O0 -S filename.c -> -S makes a .s file -> files written in assembly -> -std=c17 -- Use GCC's improvements to c11 standard -> -Wall -- all warnings -> -pedantic-errors -- enforce standards -> -g -- includes debug information, symbol table, debug mode -> -O0 -- optimization level 0; no optimizations -> -S -- compile to assembly (alternatively, use -c to compile to machine code) Assemble to Machine Code -> $ as -o filename.o filename.s -> -o filename.o -- output to filename.o Link into Executable -> gcc -o filename filename.o other.o Compiling and linking separately makes it easier to determine what caused the issue library disassembly ldd(1) - print shared object dependencies ldd prints the shared objects (shared libraries) required by each program or shared object specified on the command line. In the usual case, ldd invokes the standard dynamic linker. This causes the dynamic linker to inspect the program's dynamic dependencies and find and load the objects that satisfy those dependencies. For each dependency, ldd displays the location of the matching object and the (hexadecimal) address at which it is loaded. vdso(7) - overview of the virtual ELF dynamic shared object. The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Why does the vDSO exist at all? There are some system calls the kernel provides that user-space code ends up using frequently, to the point that such calls can dominate overall performance. The vDSO (Virtual Dynamically linked Shared Objects) is a memory area allocated in user space which exposes some kernel functionalities at user space in a safe manner. mechanisms used to accelerate certain system calls in Linux. For instance, gettimeofday is usually invoked through this mechanism. The first mechanism introduced was vsyscall, which was added as a way to execute specific system calls which do not need any real level of privilege to run in order to reduce the system call overhead. Following the previous example, all gettimeofday needs to do is to read the kernel's the current time. There are applications that call gettimeofday frequently (e.g to generate timestamps), to the point that they care about even a little bit of overhead. To address this concern, the kernel maps into user space a page containing the current time and a fast gettimeofday implementation (i.e. just a function which reads the time saved into vsyscall). Using this virtual system call, the C library can provide a fast gettimeofday which does not have the overhead introduced by the context switch between kernel space and user space usually introduced by the classic system call model INT 0x80 or SYSCALL. libc(7) – C standard library The pathname /lib/libc.so.6 (or something similar) is normally a symbolic link that points to the location of the glibc library, and executing this pathname will cause glibc to display various information about the version installed on your system ld.so(8) - dynamic linker/loader find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it. nm(1) - list symbols from object files nm displays the symbol table associated with an object, archive library of objects, or executable file. nm recognizes several different file types that may contain symbol tables: object files ending in .obj. These files may be Intel OMF (Object Module Format) files or COFF (Common Object File Format) files. library files, which normally end in .lib and contain one or more OMF or COFF files. Windows executable files, which normally end in .exe, that contain a symbol table compatible with those produced by PLINK86, the Microsoft Linker, the Watcom Linker, or the Borland Linker. A symbol table may be created from a .map file (such as those produced by Microsoft link) by using unstrip to create a symbol table compatible with PLINK86. 1. “ld” used for linking 2. “ldd main” used to check the library dependency from the executable file 3. “vdso” to make the program execute faster 4. “ld-linux…” library in charge of loading the argv, load the shared libraries 5. “nm main” to list symbols from object file (“nm /lib64/libc.so.6 | grep longjmp”) tracing system calls strace(1) - trace system calls and signals strace is a useful diagnostic, instructional, and debugging tool. System administrators, diagnosticians and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them. Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs. And programmers will find that since system calls and signals are events that happen at the user/kernel interface, a close examination of this boundary is very useful for bug isolation, sanity checking and attempting to capture race conditions. malloc(3), calloc(3), realloc(3), and free(3) Malloc The malloc() function allocates size bytes and returns a pointer to the allocated memory Normally, malloc() allocates memory from the heap, and adjusts the size of the heap as required, using sbrk(2). When allocating blocks of memory larger than MAP_THRESHOLD bytes, the glibc malloc() implementation allocates the memory as a private anonymous mapping using mmap(2). brk() and sbrk() change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory. Calling sbrk(0) can be used to determine the current program break. mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0). mmap() uses addresses outside your program's heap area, so heap fragmentation isn't a problem. mmap works by manipulating your process's page table, a data structure your CPU uses to map address spaces. The CPU will translate "virtual" addresses to "physical" ones, and does so according to the page table set up by your kernel. 131048 is the maximum ceiling for brk 131049 is the minimum for mmap critical section problem, Peterson's solution Critical section: Critical section is a part of the code responsible for changing data that must only be executed by one thread or process at a time to avoid RACE CONDITION. Each process/thread has a critical section. A solution for a critical section problem must satisfy three requirements: Mutual exclusion, progress and bounded waiting. Mutual exclusion - If there is one process executing its critical section, then no other process can executing in their critical section Progress - If no process is executing its critical section and some processes wish to enter their critical section, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely Bounded waiting - There is a limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. Two general approaches are used to handle critical sections in OS: Preemptive kernel: allows a process to be preempted and Nonpreemptive kernel: does not allows preemption, a kernel mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU Peterson’s Solution: It only uses shared memory for communication and its original

formulation only worked with two processes, but the algorithm can be generalized for more than two. Peterson's solution requires the two processes to share two data items: Whose turn it is to enter the critical section (int turn;) A flag if its ready to enter critical section (bool flag[2] = {false, false}; while (true) { // entry section flag[i] = true; turn = j; while (flag[j] && turn == j); // spin // critical section // exit section flag[i] = false; // remainder section } mutexes vs. semaphores Mutex: Mutual Exclusion lock, Software tool for ensuring mutual exclusion, accessed through two standard atomic operations: acquire and release. It can be used to protect a critical section. acquire(m) - Blocks while m is locked, then locks m release(m) - unlocks m This is a implementation not limited to pthreads. It can synchronize processes as long the atomic_flag (mut_t) is in shared memory. typedef atomic_flag mut_t; volatile mut_t mut = ATOMIC_FLAG_INIT; // false; true = locked; false = unlocked // void acquire(mut_t * m) #define acquire(m) while (atomic_flag_test_and_set(m)) // void release(mut_t * m) #define release(m) atomic_flag_clear(m) The atomic_flag_test_and_set function checks the flag value. If the value is false, then it sets the flag to true (i.e., locks the mutex). This function is guaranteed to be atomic by <stdatomic.h>. Therefore the only way to break the while loop is if the flag is false. Otherwise it will busy wait until it encounters the mutex in an unlocked state. Atomicity is important because we do not want threads to acquire the lock in between comparison. Pthreads comes with its own mutex implementation pthread_mutex_t mut; #define acquire(m) pthread_mutex_lock(m) #define release(m) pthread_mutex_unlock(m) Semaphore A semaphore is an integer variable that, apart from initialization, is accessed only through two standard atomic operations wait(s) - blocks while s<= 0, then decrements; signal(s) - increments s A semaphore is used to limit access to a resource to a specific number of threads. The initial value of the semaphore denotes the limit. Just like a bouncer in a club. Once the bar is full (the number allowed {s} equals to zero) the bouncer will only let people get in, once some of the people get out. If the initial value of s is 1, it is called a binary semaphore. A mutex is a binary semaphore. The waiting: int wait(mysem_t * s) { acquire(&s->mut); while (atomic_load(&s->val) <= 0); atomic_fetch_sub(&s->val, 1); release(&s->mut); return 0; } // wait While load and fetch_sub are guaranteed to be atomic, we can't guarantee that both lines involving these functions will execute atomically as a block, so we treat this section as its own critical section, and use a mutex to ensure mutual exclusion Linux comes with POSIX Semaphores <semaphore.h> sem_t sem; #define wait(s) sem_wait(s) #define signal(s) sem_post(s) deadlock, deadlock prevention vs. avoidance, banker's algorithm A deadlock it is a state in which each process is waiting for another process (including itself) A deadlock can arise if all of the following four conditions hold simultaneously true. Mutual exclusion: At least one resource must be held in a non-sharable mode; only one thread at a time can use the resource. If another thread request that resource, the requesting thread must wait until the resource has been released. Hold and wait: A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads. No preemption: Resources cannot be preempted; the resource can only be released voluntarily by the thread holding it. Circular wait: A set {T0, T1, ..., Tn} of waiting threads must exist such that T0 is waiting for a resource held by T1, T1 is waiting for a resource held by T2, ..., Tn−1 is waiting for a resource held by Tn, and Tn is waiting for a resource held by T0. (If a resource can be acquired by more than one thread, a cycle in the graph is a necessary but not a sufficient condition for a deadlock) Deadlock Prevention imposes restrictions on the overall system to prevent DL from ever occurring, DL Avoidance keeps track of the state of a system and take measures to avoid DL on a case-by-case basis. Prevention: To prevent Deadlock to occur, we just need to ensure that at least one of the DL condition is prevented. Disallow Mutual Exclusion - Sharable resources that does not require exclusive access, for example read only files. that several threads can be granted permission to access the file simultaneously. However we cannot prevent DL by denying mutual-exclusion because some resources are intrinsically non-sharable. Disallow Hold and Wait: We can use one protocol that requires each thread to request all its resources before it begins executing. This removes the dynamic nature of requesting resources. Alternatively a protocol that forces a thread to request resources only when it has none, it must release all of its resources before requesting new resources. This leads to low utilization. Disallow preemption: Preemption cannot generally be applied to such resources as mutex locks and semaphores, precisely the types of resources where deadlock occurs most commonly. Disallow Circular Wait: Impose a total ordering theoretically avoids the cycle, developing an ordering/hierarchy does not in itself prevent deadlock, it is up to the app developers to write programs that follow the ordering. Deadlock avoidance can have some overhead. And it has two states, safe and unsafe. The system is in a safe state if it can allocate resources to each thread up to its maximum in some order and still avoid a deadlock. Resource-Allocation-Graph, if each resource has only one instance, then we can build this graph to check if adding a request edge forms a cycle. However this is not applicable to a system with multiple instances of each resource, since a cycle its a necessary but not sufficient condition. Banker's Algorithm is a deadlock avoidance algorithm. When a new thread enters the system, it must declares maximum number of instances of each resource that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether it will leave the system in a safe state. If so, it is allocated, if not, the thread must wait until some other thread releases enough resources. paging, hierarchical paging 1. Virtual/logical vs Physical a. Virtual address: address generated by CPU that will be translated to physical address before CPU use it b. Virtual address space: set of all the virtual addresses c. Physical address: actual address in computer memory d. Physical address space: set of all physical addresses 2. MMU: memory management unit a. CPU passes virtual address b. MMU translates to physical address and access physical memory c. Relocation-based MMU d. Memory protection