CS4400 *Alex Clemmer, u0458675*

Notes

1 Control Flow

Your PC (that's program counter, not personal computer) views addresses as a series of numbers *a la* $a_0 \dots a_{n-1}$, where each a_k is the address of some corresponding instruction I_k. Transitioning from a_k to a_{k+1} is a called a *control transfer*, and continuous control transfer is called *control flow*.

In the simplest case (called *smooth transfer*), the transfer happens from one instruction to the next contiguous instruction; in reality there are other ways to do this, like jumps, calls, and returns. This allows the programs to react to changes in the state of the program.

Often we want to respond to things that are not captured by program variables, and sometimes not even related to the program at all. Stuff like incoming network packets must be stored in memory; hardware timers must be dealt with at regular intervals; and so on. So modern systems have ways of making abrupt changes in control flow.

This is called *exceptional control flow* (ECF). This happens up and down the stack: hardware can detect exceptions and transfer control to exception handlers; programs can *signal* one another, which could cause control to abruptly get sent to a signal handler; the kernel can transfer control between user processes via context switches; even an individual program can subvert traditional stack discipline by nonlocal jump to arbitrary locations in other functions.

So why study ECF? A few reasons:

- *ECF illustrates important CS concepts.* ECF is the way that operating systems implement IO and virtual memory, so knowing how they work is crucial to understanding computers.
- *ECF illustrates how programs interact with the OS.* Program requests for resources are invoked via *trap* or *system call*. This happens whe programs request memory, reading to network, creating new processes, etc. System call mechanisms go a long way in learning why things work the way they do.
- *ECF helps write programs.* Doing things like creating processes, sending signals, and waiting for processes to terminate are super useful in awesome applications like shells and servers.
- *ECF helps illustrate concurrency.* Causing a program interrupt via exception, using simultaneously-executing threads, and signal handlers that interrup program use are all examples of concurrency.
- *ECF helps you understand how sofware exceptions work.* Exceptions happen all the time in C++ and Java, and you can do nonlocal jumps from within the application via application-level ECF, and in C using `setjmp` and *longjmp*.

1.1 Exceptions

An *exception* is an abrupt change in control flow not encoded by program state, that is caused by some change in the *processor's* state. That is, ECF is implemented partly at the hardware level and partly at the software level, and though the details are different across systems, the concepts are pretty similar throughout.

We begin by imagining that we are executing instruction I_{curr}. Then some significant state changes inside the processor—maybe it's in direct relation to I_{curr}, like a page fault or a divide-by-zero error; or maybe this is not at all the case, as would be true of IO completion or system timer alarm.

Once the processor detects that this has happened, it does and indirect procedure call through the OS's *exception table*, at which point it is routed to an OS subroutine that is designed to handle this error, called an *exception handler*. This is a hardware exception, different from a software exception—this is not able to be caught at the software level, and one of a few things happens:

- We return to the current instruction I_{curr}.
- We return control the I_{next} instead of I_{curr}.
- We terminate the process.

1.1.1 Exception Handling

Each possible exception gets a nonnegative integer *exception number*, some of which are specified by the hardware, and some of which are specified directly by the *kernel*, or the memory-resident component of the OS. The kernel specifies things like signals form IO devices and system calls, where the hardware specifies things like divide-by-zero errors, arithmetic errors, and memory access errors.

At boot time, the OS initializes the *exception table*, which is a jump table that contains addresses of the exception handlers, with the kth entry being the address to the handler of the signal whose number is k. If a hardware exception happens at run time, then the program looks in the CPU for the *exception table base register*, which contains the address of the start of the exception table, and then jumps to the kth entry in that table, which contains the address of the appropriate handler. This is an indirect procedure call through entry k of the exception table.

So how is this different from a procedure call? First off, although here the return address is saved, the return address may either be I_{curr} or I_{next}, depending on the sort of exception that has occurred. Second, we push some information onto the stack that will be required to restore the program later. In IA32, we push things like EFLAGS to the stack, which determines gives us information about the sort of error caused. Third, we push either to the stack or the kernel's stack, depending on where control is being transferred to the kernel. Fourth, the exception handler is run in *kernel mode*, which means it has access to all system resources.

Everything post-exception happens at the software level, and after the handler is done processing, it is possible to optionally relay control back to the interrupted process, which is done via popping everying from stack into registers. When this happens, we switch from kernel mode to *user mode*.

1.1.2 Classes of Exceptions

Exceptions, if you didn't know, are divided into four classes: *interrupts*, *traps*, *aborts*, and *faults*.

Interrupts

Interrupts are a completely asynchronous excption, as they are not prompted by any particular instruction. They are handled by *interrupt handlers*, and are usually caused by things like network adapters, disk contorllers and timer chips.

This type of exception works by tripping a pin on the processor chip and writing an exception number on the system bus. When the processor finishes executing the current instruction, it notices the pin has been flipped, looks at the exception number written to system buss, and routes to the appropriate interrupt handler via the exception table base register, through which we are routed to the exception table.

When all this is done, we transfer control back to I_{next}. The program continues executing as though nothing at all happened.

Traps and System Calls

A *trap* is a *synchronous* exception, because it is caused by the *intentional* execution of an instruction that specifically causes a trap. Trap handlers route control back to I_{next}, much like interrupt handlers.

The most common and important use of traps is that they act as an interface to *system calls*. This most often happens when a program needs some service n provided by the kernel (*e.g.*, `fork` or `exit`). The *processor* usually allows a special instruction like `syscall` n; at the kernel level, control flow is routed to the exception handler that handles `syscall`, at which point the exception handler decodes the argument n and routes it to the proper kernel routine.

How is this different form a normal function call? One crucial difference: system procedures like `fork` operate in kernel space, not user space. They are given a kernel-defined stack and access to system resources.

Faults

Faults are another type of synchronous exception, which is to say, they are caused directly by executing an instruction. Faults result from errors that are believed to be recoverable, like page faults.

Typically, a *faulting instruction* triggers an exception, at which point the exception handler will attempt to recover form the error. If it can, then we recover and eventually return to I_{curr}; if we cannot, we abort the process.

Aborts

Aborts are also synchronous exceptions, in that they are caused by the excecution of a specific instruction. However, unlike previous exceptions, abort never returns. Usually caused by something like DRAM/SRAM parity errors, aborts usually cause process termination.

1.2 Exceptions in Linux/IA32 Systems

Intel defines exceptions numbered 0 to 31, with 32 to 255 left to interrupts and traps defined by the operating system. Examples of the former include divide errors, page faults, machine faults, and so on. Examples of the latter include syscalls.

Linux/IA32 Faults and Aborts

Divide error is caused when some application divides by zero, or when the result is too big for a destination operand. Unix doesn't recover from errors, and instead simply aborts the program.

General protection fault typically happens when you try to access memory that is not defined by virtual memory, or when you try to write to read-only text segments of data. Linux reports this as a segfault.

Page fault instruction not in main memory; map to appropriate page, pull from disk, return to I_{curr}.

Linux/IA32 System Calls

There are hundreds of these, used for reading and writing files, printing to stdout, creating processes, etc. You could call the `syscall` function, but usually there are system level functions like `fork` that you can call for convenience's sake.

2 Processes

A process is *an instance of a program in execution.* Every program is run in the *context* of some process. This context is the state that the program needs to run—things like the code, the data stored in memory, the stack, general purpose registers, the PC, environment variables, and open file descriptors.

Processes are characterized primarily by a couple of things: *independent control flow logic* that gives us the illusion that the program has exclusive use of the processor; private address space that provides the illusion that our program is the only one in memory.

2.1 Logical Control Flow

Logical control flow is the sequence of addresses you would see in the PC as it executes your program—in other words, instructions that are in your programs executable or shared libraries that it links to.

While it looks like logical control flow is sequential on the whole, it's actually interrupted and interleved constantly, and the CPU is often weaving between dozens of them at once. A program will execute for some time before being *preempted*, or temporarily suspended as another process takes its turn.

2.2 Concurrent Flows

If we have to logical control flows operating that overlap in time, those flows are said to be *concurrent*, with the notion of processes taking turn being called *multitasking.* These concurrent flows are made up of *time slices*, which is the time period that a process executes a portion of its flow.

Notice that this notion does not include a notion of cores on a chip. We might think of two processes operating on different cores as parallel, but it is not strictly speaking always necessary to do so.

2.3 Private Address Space

Each process gets its own address space. In general other processes can't access the memory from another process's private address space.

2.3.1 User and Kernel Modes

The *user* abstraction is actually provided at the processor level: when the *mode bit* is tripped, the processor has access to all memory and can execute any instruction.

When the mode bit is not set, we are in *user mode*, and there are distinct limitations. Users are not allowed to execute what are called *priviledged instructions* that do things like halt the processor, change the mode bit, or initiate IO. It is also not allowed to access kernel data, kernel instructions, or the address space of the kernel. Instead, they must get to the kernel via some sort of exception, such as the previously-mentioned *trap* technique.

Some kernel data is actually available, through `/proc`, which allows you to do things like look at the CPU type. Newer systems also have `/sys`, which exposes information about the system bus.

2.3.2 Context Switches

Multitasking is implemented via a higher-level ECF called *context switching.* The kernel maintains a *context* for all current processes. Context is basically the state that needs to persist between time slices so that the

program doesn't fail. It includes things like variables, stack, kernel stack, kernel data structures (*e.g.*, page table, process table and file table), and so on.

At any time the kernel can preempt the process; in general this decision to preempt a process so that a previously-preempted process can run is known as *scheduling*, and is usually handled with code inside the kernel called the *scheduler*. The actual transfer of control from one process to another is called a *context switch*. In order to perform a context switch the kernel:

- Saves the context of the current program.
- Restores the saved context of another program.
- Passes the other program control.

This can actually happen even during system calls, especially calls like `read`, which usually blocks. But even if a process does not block, we can still context switch.

Context switches can actually occur as a result of interrupts, for example, as a result of a clock interrupt, the kernel might decide that some process has been running long enough, and it simply switches.

2.3.3 System Call Error Handling

Of course it is always possible for calls to things like `fork` to fail. In the event of failure, the error code is written to the global variable `errno`, and the function returns -1. We should probably always check for errors, but most people do not. One example of checking errors includes the following fork:

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

2.4 Process Control

The Unix environment provides a nice systems interface. One example of this is the build-in system calls that are wrapped in functions. Let's look at some of them now:

2.4.1 Obtaining Process IDs

Two functions are useful here: `getpid` and `getppid`, which return the process ID (PID) of the current and parent process, respectively. Note that every process is a *unique positive number*, which remember, means that it's *not* zero. This number is wrapped as a `pid_t`, which in `types.h` is really an `int`.

2.4.2 Creating and Terminating Processes

Here's where it gets a bit interesting. We typically think of processes in one of three states. If a process is *running*, then it is either executing currently, or it is waiting to run, eventually being scheduled by the kernel. If a process is *suspended*, then it is stopped and will not be scheduled until it gets a SIGCONT signal—a software interrupt, by the way, that we will see shortly. It is put in this state by signals like SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU. The last possible state is the *terminated* state. This is what it sounds like. A process could terminate for a number of reasons. It might have been terminated by a hardware exception, it may have finished, and it may have called `exit`.

The `exit` function, by the way, will terminate some process with an exit of `status`.

The issue of creating is slightly trickier. One way to do this is with `fork`, which creates a child process that is almost, but not quite identical to the parent process. In particular, the

3 Virtual Memory

VM operates primarily as an abstraction of main memory. It's annoying and error-prone to deal with raw hardware addresses, and it is therefore useful to have a virtual interface that allows us to map things to memory. In short, we tell the kernel where in virtual memory to put things, and it maps that to a real address. There are a couple fundamental things to note here:

1. **VM is treats MM as cache for disk.** We don't want to deal with disk at all, actually, and the abstraction of VM allows us to imagine that there is just *one* type of memory. Swapping memory into and out of disk is something handled at a lower level.
2. **VM simplifies memory management.** Each process gets a uniform address space, rather than a fragged-up awful memory space, which is what memory really looks like.
3. **VM reduces memory errors.** With VM, it's harder to write over someone else's memory.
4. **VM works silently.** The programmer does not have to implement this. Which is a great advantage.

3.1 Main Memory (MM)

We can view MM as an array of M contiguous byte-sized cells, each of which has a unique *physical address* (PA). When we use physical addressing, we are using the actual PA to access the memory.

In order to fetch memory using this scheme, the CPU will pass an address x to the main memory controller via the memory bus, which will then fetch the word starting at PA x. This memory is then sent back to the CPU, which will store it in a register somewhere.

3.2 Virtual Addressing

MM accessing these days is usually not used for modern computers for general-purpose computing (this story is somewhat different for embedded devices). Instead of using PAs, we use *virtual addresses* (VAs).

In order to fetch memory using this scheme, the CPU generates a VA y, which is converted into a PA before being passed to main memory. This step is somewhat involved—it requires a close, concerted effort between the CPU and the OS in order to function correctly. Specifically, this step uses the *memory management unit* (MMU), which is dedicated CPU hardware designed to translate VAs into PAs, using a lookup table designed in memory.

3.3 Address Space

So what's the playing field we're looking at? For starters, generically an **address space** is a range of discrete addresses—usually a set of integers > 0. If the integers are consecutive, then this address space is *linear*. Note that each byte has both a PA and a VA.

3.3.1 Virtual Address Space

The *virtual address space* is, in our case, a set of $N = 2^n$ virtual addresses, with n being the number of bits you need to represent the largest address in binary. A typical n in modern computing would be 32 or 64.

3.3.2 Physical Address Space

The *physical address space* corresponds to the $M = 2^m$ bytes of physical memory that the system has.

3.4 Main Memory as Cache

Your VM corresponds to N byte-sized cells stored on disk, each of which has a VA that is an index into this array. That is, VM is an abstraction of all of disk memory, with MM swapping occurring *transparently* to the user.

This N-byte array is actually partitioned into blocks, commonly called **pages**. The **page size** is $P = 2^p$.

3.4.1 DRAM Cache Organization

DRAM is about 10 times slower than SRAM; disk is about 100,000 times slower than DRAM; and finally, seeking to a disk sector is about 100,000 times slower than reading successive bytes after you've reached that sector.

The organization of DRAM caches is largely driven by this prohibitive miss penalty (DRAM caches are fully associative), and the expense of accessing the first disk sector byte (vitual pages are 4-8 KB).

3.4.2 Page Table

The *page table*, a data structure stored in main memory, maps VAs to PAs. When the CPU requests a specific VA, the address translation hardware reads the page table and transfers data between disk and DRAM.

A page table basically consists of an array of *page table entries*. Every VP has a PTE at a fixed offset in in the page table. The PTE straightforwardly consists of:

1. A **valid bit**, indicating whether the VP is currently cached in DRAM.
2. A **k-bit address field**. When the valid bit = 1, corresponds to the starting address of the corresponding PP in DRAM. When the valid bit = 0, a `NULL` address indicates no allocated VP, while a non-`NULL` address indicates the start of the VP on disk.

3.4.3 Page Hits

When the CPU requests memory that's in a VP whose PTE has a valid bit = 1, the we have a *page hit*. The MMU will locate the correct PTE and check the valid bit. It will then take the k-bit address field, which corresponse to the PP, and then pull that out of DRAM. Concretely, this is what happens:

1. CPU generates a VA and sends to the MMU.
2. MMU generates the address for the PTE and requests it from memory.

3. Memory returns the PTE to the MMU.
4. The address is valid, so the MMU requests the PA starting at the k-bit address.
5. Memory returns memory at said address.

3.4.4 Page Fault

When the MMU looks at a PTE whose valid bit is set to 0, this indicates a page fault. If the k-bit address is non-`NULL`, we fetch the VP from disk, and map that into DRAM. If DRAM is already full, we might select a victim page. This is a fault, so we restart the instruction from scratch, but this time, it's a page hit. All is right with the world. More concretely, this is what happens:

1. CPU generates a VA and sends to the MMU.
2. MMU generates the address for the PTE and requests it from memory.
3. Memory returns the PTE to the MMU.
4. The valid bit = 0; MMU causes page fault exception, transferring control to the OS.
5. The fault handler identified the so-called victim page, paging out to disk if we need to write-back.
6. We create a new PTE and update it in memory.
7. Return control to the original process, restarting at the faulting instruction. It's now a page hit.

3.5 VM Benefits, In More Detail

As you can now probably see, this allows us some pretty great benefits w.r.t. things like linking. Because each process uses the same basic format for its memory image, without regard to where it is physically. You can map different VPs in different processes to the same PP, which allows different processes to share data and/or code. It also makes allocation simpler, since we really only have to worry about allocating contiguous VPs, rather than actual PPs.

But there are deeper benefits too. We want the OS to control memory access, because a user *should* be prevented from writing to its read-only data, or modifying kernel data, or modifying private data of other processes. You can do this by adding permission bits to the PTE, which when violated trigger a protection fault, usually a segfault.

3.6 Translation Lookaside Buffer

The TLB is a small cache of PTEs in the MMU. The thinking goes, if we store them in the L1 cache, then it will cost a small amount of time to get; however, if it's not, it may take anywhere from tens to thousands of cycles (*i.e.*, we'd have to go either to L2 or MM).

The TLB is virtually addressed, with each block holding a PTE. It typically has a high degree of associativity, $T = 2^t$ sets.

4 Dynamic Memory

The *heap* begins after the `.bss` section, and grows upward. The kernel maintains a variable, `brk`, that points at the top of this stack. A *dynamic memory allocator* maintains the heap as a collection of variously-sized blocks. Each of these blocks is either:

1. *Allocated:* explicitly reserved for some application, and remaining so until freed.
2. *Free:* explicitly free, and remaining so until explicitly allocated.

There are fundamentally two types of allocators:

1. *Explicit allocators*, which require the application to free all blocks manually.
2. *Implicit allocators:*, which require the allocator to clean up after itself with garbage collection.

4.1 `malloc`, an Explicit Allocator

```
void *malloc(size_t size);
```

`malloc` returns a pointer to a block of *at least* `size` bytes, which is suitable for any sort of data storage you see fit. If `malloc` encounters a problem, it returns `NULL`, and sets `errno` appropriately. Reallocating a previously-allocated block requires:

```
void *realloc(void *ptr, size_t size);
```

4.1.1 The `sbrk` Function

```
void *sbrk(int incr);
```

`sbrk` increments the value of `brk` by `incr`. If this is successful, it returns the old value of `brk`; if *not* successful, we return -1, and `errno` is set to `ENOMEM`.

Neat hack of the day: set `incr` $= 0$, which will just return our current `brk`.

4.1.2 The `free` Function

```
void free(void *ptr);
```

This function simply frees the block indicated by `ptr`. Note that `free` is undefined if `ptr` does not point to the beginning of an allocated block. Also there is no indication if something went wrong, since `free` does not return anything.

4.2 Fragmentation

One problem with allocators in general is fragmentation. There are two main sorts of fragmentation:

1. *Internal fragmentation*, which occurs when an allocated block is actually larger than the payload.
2. *External fragmentation*, which occurs when there is enough memory to accomodate a request, but it is not *contiguous*.

4.3 Implemention Issues

1. *Free block organization*, how do you keep track of free blocks?
2. *Placement*, how do you choose among free blocks to place your new memory?
3. *Splitting*, what do you do with the rest of a free block once you've put a less-than-optimal block inside of it?
4. *Coalescing*, what do you do with a block that has just been freed?

4.4 Block Format

We want to distinguish between blocks that are allocated and blocks that are not; we do so in this case by embedding the information directly into the block. Every block has a *payload*, a *header*, and a *footer*. `malloc` returns a pointer at the beginning of the payload. The header contains the size of the payload as a multiple of 8—the bottom 3 bits are metadata, with the least significant bit denoting whether that block is allocated or not.

5 Networks

Web, email, and X windows are all examples of network applications. All network applications are based, more or less, on the same model:

5.1 Client-Server Model

Each app consists of 1 server process and ≥ 1 client process(es). Servers manage some set of resources, and provides some service for clients by manipulating that resource. FTP servers store files for clients; email servers manage a spool file that it reads/updates.

Most clients and servers operate on different hosts, and communicate using networks, but to hosts, network communications are usually just another IO device. This is a convenient abstraction, because networks are usually hierarchical clusters, organized by geographical proximity, with LAN at the lowest level.

5.2 Ethernet

Ethernet is a type of LAN. Each host is connected to the hub by the Ethernet cord, and when anything is written on any of the ports, the hub copies it to *all* of the other ports. So if anything is written on the network, *everyone* sees it.

Each Ethernet adapter has a unique 48-bit *address* stored in non-volatile memory. A host can send a *frame* to any other host on the segment; a frame contains a header and a payload, which notes the source, destination, and length of the frame. Important to note is that every host actually sees the frame, although the destination host is the only one that is supposed to read it.

Many of these Ethernet segments can be connected into larger LANs via wires and *bridges*. Unlike hubs, bridges copy frames from one port to another one on the basis of necessity: if the message is going to a host on the same segment, then the frame is thrown away. If it's not, then it's copied to the appropriate port.

5.3 The Internet

The internet consists of a bunch of LANs glued together mainly by special computers called *routers*. The routers each have a port for every network they're connected to. Different networks usually don't care if other networks are all operating using disparate and awful technology, because protocol software running on the hosts and the router work together to iron these differences out. These protocols help the disparate robots get along.

5.4 Protocols

A good protocol absolutely needs two things:

1. *A naming scheme*, because most LAN technology have completely different and completely incompatible ways of assigning names to different hosts. The internet solves this problem with a uniform host address format.
2. *A delibery mechanism*, because different LAN technology have different ways of encoding the same thing. The internet solves this problem by dividing things up into chunks called packets, which consist of a *header* (including size and src/dest addresses), as well as the *payload*.

6 Concurrency

One way to implement concurrency is to use *processes*. One example is a stupid server: we fork a new child process every time we accept a connection from a client, causing there to be multiple threads executing concurrently. Is this good or bad?

1. **Good:** Parents and children have good sharing, including file tables, so socket descriptors at the time of fork of the parent are shared with children.
2. **Bad:** Memory addresses are different, so you can't really share state information as easily as you'd like. Instead, you must use interprocess communication (IPC). Yikes

Let's do something else.

6.1 Threads

A thread is a logical flow that runs in the context of a process. Threads in a process share virtual address space, but also have their own context, including their own stack, stack pointer, and Thread ID (TID). This makes sense—we share the same heap, but each function invocation between threads should *not* be shared.

Life for threads begin with a *main* thread, which can spawn a bunch of *peer* threads, each of which operates concurrently w.r.t. the other threads.

There are some good things about the thread model:

1. Switching between threads is faster, since a thread context is *much* smaller than a process context.
2. Threads are not as rigidly structured: all threads in a process are "pooled"; any thread may kill or join another thread; every peer writes to the same data.

```
/* Pthreads is a standard interface for manipulating threads from C
   programs. */
#include "csapp.h"
void* thread(void* vargp);
/* main thread */
int main() {
    pthread_t tid;   /* thread ID of peer thread */
    /* create peer thread */
    Pthread_create(&tid, NULL, thread, NULL);
    /*--Now, main thread and peer thread are running concurrently.--*/
    /* wait for peer thread to terminate */
    Pthread_join(tid, NULL);
    /* terminate all threads */
    exit(0);
}
/* The code and local data for a thread are encapsulated in a thread
   routine.  Each thread routine takes as input a single generic
pointer and returns a generic pointer. */
void* thread(void* vargp) {
    printf("Hello, world!\n");
    return NULL;   /* terminate peer thread */
}
```

6.1.1 Creating Threads

```
typedef void* (func)(void *);
int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
```

This function creates a new thread and runs function `f` in the context of this new thread, with input `arg`. You *could* change default thread attributes with `attr`, but normally you wouldn't. We also return `tid`, the ID of this new thread.

6.1.2 Terminating Threads

A thread can return in a number of ways: (1) control of the top-level function returns, (2) the thread calls `exit`, and (3) another thread calls `pthread_cancel` with the current thread's ID.

6.1.3 Reaping Threads

```
int pthread_join(pthread_t tid, void **thread_return);
```

This function blocks until `tid` terminates, assigns the `void*` return value to the `thread_return`, and reaps all resources held by thread `tid`. Unlike `wait_pid`, this thread can only wait for a specific thread to terminate.

6.1.4 Detached Threads

By default, every thread is *joinable*, which means that it can be reaped and killed by other threads. If you know what you're doing, it's possible to *detach* that thread, which makes it impossible for peers to reap or kill it, although at return time, its resources are still freed.

To avoid memory leaks, *you should aways reap or free threads. Always!*

6.1.5 Avoiding Races

FAIL:

```
int connfd = Accept(...);
Pthread_create(&tid, NULL, thread, &connfd);
...
void* thread(void* vargp) {
    int connfd = *((int*)vargp);
...
}
```

If main thread's `Accept` occurs first, then this is incorrect. If peer thread's assignment happens first, then this is correct.

This is something to keep in mind: while each thread has its on context (PC, SP, stack, TID, general-purpose registers, etc.), each thread shares *the rest of process context* with other threads. Files, code/heap, and shared libraries are all shared among peers. Further, it is *not* true universally that thread stacks are accessed independently by their respective threads. Local `static` variables, for example, may be accessed by any thread.

1. *Global variables* are any variables that are declared outside of a function. **Any thread** may alter these at will.
2. *Local automatic variables* are any variables that are on the stack, and local to a function (*i.e.*, not declared as `static`). **One thread** may alter these.
3. *Local static variables* are local variables declared with `static`. **One instance** for all threads.

7 Semaphores

A semaphore is a global variable $s \geq 0$ that has exactly two operations, each of which occurs **atomically**:

1. $P(s)$: $s--$ if $s \neq 0$ (*i.e.*, if s is not "locked", note that this entire comparison and decrement happens *atomically*); if $s = 0$, we suspend the process until it becomes nonzero, and afterwards $s--$ and return.
2. $V(s)$: $s++$ unconditionally, then check to see if any operations are blocking in a P operation, and if they are, restart exactly one of them.

In other words, $P(s)$ locks a resource and $V(s)$ unlocks it. After unlocking it, you restart exactly one of these things.

7.1 Binary Semaphores

Semaphores s is initially set to 1, which is to say, it is *unlocked*. So was surround the dangerous code with $P(s)$ and $V(s)$ operations.

7.2 Semaphores in POSIXland

The following implement the functionality described above:

```
int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *sem);  /* P(s) */
int sem_post(sem_t *sem);  /* V(s) */

void P(sem_t *sem);  /* wrapper */
void V(sem_t *sem);  /* wrapper */
```

Here is an example of use:

```
sem_t mutex;  /* semaphore to sync cnt access */
sem_init(&mutex, 0, 1);  /* init mutex */
...
for (i = 0; i < NITERS; i++) {
    P(&mutex);  /* protect shared */
    cnt++;  /* increment */
    V(&mutex);  /* unlock */
}
```

8 Producer-Consumer Problem

Also called the *bounded buffer* problem, refers to a multi-process synchronization where (broadly speaking) we have a buffer that is filled one slot at a time by one process, and emptied one slot at a time. The trick is to not remove any slots from an empty buffer, or add slots from a full one!

More formally, let's say that the producer and consumer threads share n slots; since the producer thread adds things to the buffer, and the consumer removes them, what we must do is assure *mutually-exclusive access*, and that the relevant full/empty constraint is fulfilled.

```
typedef struct {
    int *buf;  /* Buffer array */
    int n;  /* Max # of slots */
    int front;  /* buf[(front + 1) % n] is 1st item */
    int rear;  /* buf[rear % n] is last item */
    sem_t mutex;  /* protects accesses to buf */
    sem_t slots;  /* counts available slots */
    sem_t items;  /* counts available items */
} sbuf_t;

/* Create an empty, bounded, shared FIFO buffer with n slots */
void sbuf_init(sbuf_t* sp, int n) {
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;                        /* Buffer holds max of n items */
    sp->front = sp->rear = 0;      /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1);      /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n);  /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0);   /* Initially, buf has 0 data items */
}
```

```
/* Clean up buffer sp */
void sbuf_deinit(sbuf_t* sp) { Free(sp->buf); }
/* Insert item onto the rear of shared buffer sp */
void sbuf_insert(sbuf_t* sp, int item) {
    P(&sp->slots);                        /* Wait for available slot */
    P(&sp->mutex);                              /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item;       /* Insert the item */
    V(&sp->mutex);                             /* Unlock the buffer */
    V(&sp->items);                        /* Announce available item */
}
/* Remove and return the first item from buffer sp */
int sbuf_remove(sbuf_t* sp) {
    int item;
    P(&sp->items);                        /* Wait for available item */
    P(&sp->mutex);                              /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)];      /* Remove the item */
    V(&sp->mutex);                             /* Unlock the buffer */
    V(&sp->slots);                        /* Announce available slot */
    return item;
}
```

9 Prethreading

One problem with the echo server we wrote earlier is that we created a thread once for every connection. That's expensive. A better solution is to simply create one server thread and $n-1$ worker threads. Basically, the server then works as so:

1. Server accepts requests from clients, placing each socket descriptor into the bounded buffer.
2. The worker threads repeatedly remove descriptors from the buffer, serving each, and then waiting for the next descriptor.

```
/* echoservert_pre.c  a prethreaded concurrent echo server */
sbuf_t sbuf; /* shared buffer of connected descriptors */
int main(int argc, char* argv[]) {
    int i, listenfd, connfd, port, clientlen=sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    pthread_t tid;
    if (argc != 2)
        /* ERROR, QUIT */
        port = atoi(argv[1]);
    sbuf_init(&sbuf, 16);
    listenfd = Open_listenfd(port);
    for(i = 0; i < 4; i++)  /* Create worker threads */
        Pthread_create(&tid, NULL, thread, NULL);
    while(1) {
        connfd = Accept(listenfd, (SA*) &clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
    }
}
void* thread(void* vargp) {
```

```
    Pthread_detach(pthread_self());
    while(1) {
        int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
        echo_cnt(connfd);                /* Service client */
        Close(connfd);
    }
}

/* A thread-safe version of echo that counts the total number
   of bytes received from clients. */
static int byte_cnt;  /* byte counter */
static sem_t mutex;   /* and the mutex that protects it */
static void init_echo_cnt(void) {
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
void echo_cnt(int connfd) {
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;
    Pthread_once(&once, init_echo_cnt);
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        P(&mutex);
        byte_cnt += n;
        printf("thread %d received %d (%d total) bytes on fd %d\n",
            (int) pthread_self(), n, byte_cnt, connfd);
        V(&mutex);
        Rio_writen(connfd, buf, n);
    }
}
```

10 Other Concurrency Issues

So far we've basically covered a simple subset of the techniques we have for slaying the dragons of concurrency—things like mutual exclusion and producer-consumer synchronization. But the problems of concurrency extend beyond these simple models. Let's cover some of the common issues.

10.1 Thread Safety

A function is said to be thread-safe *if and only if* it always produces the correct result when called from multiple concurrent threads. A function that does not satisfy this quantity is said to be *thread-unsafe*. There are four types of thread-unsafe functions:

1. Those that do no protect shared variables.
2. Those that share state across multiple invocations.
3. Those that return pointer to static variables.
4. Those that call thread-unsafe functions.

10.1.1 Shared Variables

```
void count(void *arg)
{
    int i;
    for (i = 0; i < NITERS; i++)
        cnt++;
    return NULL;
}
```

In this context, the internal use of a variable that is shared will likely ruin our ability to produce the correct result! In this case we need to protect `cnt` with synchronization operations. The *good news* is that the interface remains unchanged; the *bad news* is that the function will run slower, since it will use some sort of control system.

10.1.2 Shared State over Multiple Invocations

```
unsigned int next = 1;
/* rand - return  pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int) (next / 65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

In this case, we must alter the program to pass state information in when calling the function—which in other words means we need to change the function signature as well as the callers.

10.1.3 Returns Pointer to Static

If you return a static variable, it may be (silently) overwritten in other parts of the program. Obviously this is bad! There are two ways to possibly fix this:

1. Pass in as an argument the address at which you'd like to store the data; this, though, requires changes to calling code.
2. *Lock and copy* – wrap all data access in mutices, each of which is associated with a specific function. You: (1) allocate memory dynamically, (2) lock the mutex, (3) call the unsafe function, (4) copy the result into this memory, and (5) unlock the mutex.

Looks like this:

```
struct hostent* gethostbyname_ts(char* hostname) {
    struct hostent *sharedp, *unsharedp;
    unsharedp = Malloc(sizeof(struct hostent)); /* dyn mem */
```

```
    P(&mutex);                                /* lock mutex */
    sharedp = gethostbyname(hostname); /* thread-unsafe fn */
    *unsharedp = *sharedp;       /* copy to private struct */
    V(&mutex);                               /* unlock mutex */
    return unsharedp;
}
```

10.1.4 Calls Thread-Unsafe Function

If you have some function f and a thread-unsafe function g, then f may or may not also be thread-unsafe. If g shares memory across invocations, you must simply rewrite it. If g does not protect shared variables or returns a static variable, f still might be safe. See above for strategies of dealing with this.

11 Reentrancy

Reentrant functions do not access any shared data when called by multiple threads, making reentrant functions a proper subset of threadsafe functions. They're also usually faster, since there are no synchronizations operations.

The only way to convert a class 2 thread-unsafe function into a thread-safe on is to make it reentrant:

```
/* rand_r - a reentrant pseudo-random integer generator */
int rand_r(unsigned int* nextp) {
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp / 65536) % 32768;
}
```

11.1 Types of Reentrancy

1. *Explicitly reentrant* functions recieve all data by-value (rather than by-reference), and all data references occur to automatic stack variables.
2. *Implicitly reentrant* functions accept pointers, but it is up to the caller to be careful not to pass in shared data; other than that, they are exactly the same (with automatic variables, etc.). `rand_r` above is an example of this.

12 Race

A *race* happens when the correctness of some program depends on control at one thread reaching some point x before another thread reaches y. This example **has a race**:

```
void* thread(void* vargp);
int main() {
    pthread_t tid[4];
    int i;
    for(i = 0; i < 4; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for(i = 0; i < 4; i++)
        Pthread_join(tid[i], NULL);
```

```
    exit(0);
}
/* thread routine */
void* thread(void* vargp) {
    int myid = *((int*)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

This example **has no race**:

```
void* thread(void* vargp);
int main() {
    pthread_t tid[4];
    int i, *ptr;
    for(i = 0; i < 4; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
        /* why not call free here? */
    }
    for(i = 0; i < 4; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}
/* thread routine */
void* thread(void* vargp) {
    int myid = *((int*)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

13 Deadlock

Occurs when a collection of threads are blocked, all waiting for a condition that will never be true. This is often difficult to predict in a program, for the reason that some trajectories will avoid it, while others will not. However, if we have a pair of binary semaphores, it is the case that we can prove their correctness:

> Given pair of mutexes (s, t) in a program, if each thread holds both of them and *always* locks s and t in the same order, the program is deadlock-free.