Concurrent Programming

Tuesday, December 3, 2019 4:14 PM

Kernel: the higher-level program in the operating system that controls startup (after bootloader) and which programs run when

Concurrent: runs multiple tasks at a single time

Useful on literally every level, from computing values to running Risk of Rain and Don't Starve at the same time. As such, there's lots of different levels of concurrency, and different (but similar) ways to implement them

Processes

Simplest way is to use Processes

Servers that have to handle multiple users simultaneously: Accept connection requests using the parent, then fork and have a child process service a client

Process = process context + code, data, and stack
Process context:

- Program context
  - Data registers, condition codes, program counter, etc.
- Kernel context
  - virtual memory structs
  - brk pointer

Process = Thread + code, data, and kernel context

Creating processes:

parent process creates a new running child process by calling fork

int fork(void)
returns twice when called once - but returns to different processes

returns 0 to the child process, child's PID to parent process

Child is almost identical, but has a different PID, and an identical but separate copy of the parent's virtual address space

```
pid_t pid;
int x = 1;

pid = Fork();
if (pid == 0) {   //child
 printf("child: x=%d\n", ++x);
exit(0);
}
// parent
printf("parent:x=%d\n", --x);
```

OUTPUT:
parent: x = 0
child: x = 2

waitpid() - waits until the specified child process has terminated

Reaping Child processes

When process terminates, it still consumes system resources - exit status, various OS tables

Called a "zombie" - half alive and half dead

zombies stick around as long as someone could use their exit behavior or resources they pass. for example, if a child threat 'exit(1)'s, maybe the parent could use that exit status, so the child must stay around as a zombie

int execv(const char *path, char *const argv[]) - loads and runs programs

- Replaces the currently running process with the specified new process

int execve(char *filename, char *argv[], char *envp[])

- loads and runs the file *filename* with argument list *argv*, and environment variable list *envp*
- *environment variables*

called once and never returns

Processes share file tables, but contain fully separate address spaces - this is good most of the time (you can't accidentally overwrite someone else's virtual memory), but can make sharing information between them cumbersome and difficult.

+ can't accidentally fuck up memory - much easier to work with
- sharing information between processes is difficult
- lots of overhead for processes, slower than other methods

Multiplexing - create our own logical flows and use I/O multiplexing to schedule them. Only one process, but the flows shape the address space

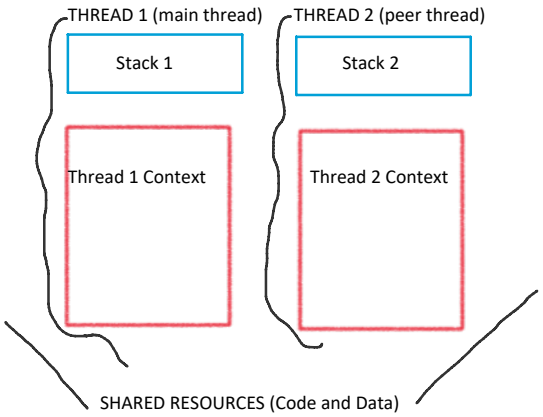
Threading

another method of utilizing concurrent programming
a hybrid of multiplexing and processes

Process = thread + code, data, kernel context
Thread = process - (code, data, and kernel context)

So just a Stack, and the Thread Context

A thread consists of:
- Its own Logical Control Flow
- Its own Thread ID (TID)
- Its own stack for local variables
    - Not protected from other threads
- A SHARED code, data, and kernel context



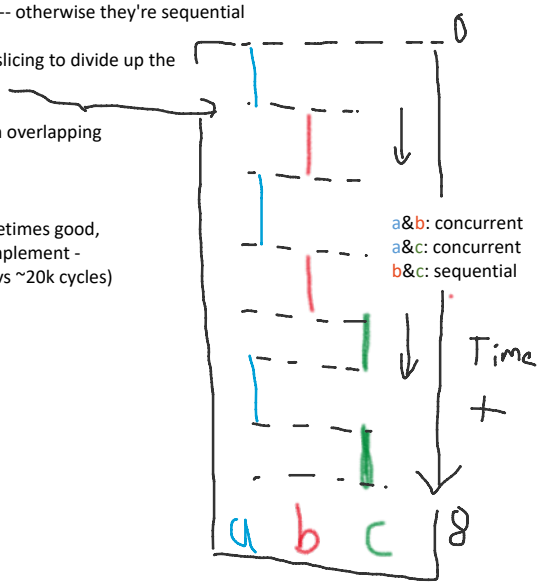
Thread Context includes:
- Data Registers
- Condition codes
- Stack Pointer (SP)
- Program Counter (PC)

-Multiple threads can be associated with a single process - process is a higher level of control
Threads associated with a single process form a pool of peers - the homies

-Threads are Concurrent if their flows overlap in time -- otherwise they're sequential

-Single-Core processor simulated parallelism by time-slicing to divide up the program - like what we see in A, B, and C over there

-Multi-Core processors can have true parallelism, with overlapping program execution times.



In addition to sharing more data than processes (sometimes good, sometimes bad), threads are somewhat cheaper to implement - they're roughly half as expensive as processes. (~10k vs ~20k cycles)

Posix Threads (Pthreads)

~60 functions used to create and manipulate threads

pthread_create() - create a thread

pthread_join() - join threads

pthread_self() - returns the thread ID

Terminating Threads
- pthread_cancel(tid…) - terminates a thread
- pthread_exit() - terminates this thread
- exit() - terminates all threads
- RET - terminates current thread

-When a thread exits, there's spare refuse (its stack in particular) that has to be cleaned up, much like zombie processes. For this, use pthread_join()

Threads are always either *joinable* or *detached*
Joinable: able to be joined and manually reaped (Default state!)
Detached: Can't be reaped/killed by other threads. Automatically terminates and reaps itself when it finishes.
- To make detached: pthread_detach(pthread_self)
- Can be convenient and save memory if you don't want/have to manually join threads

Thread Summary:
+ Easy to share data
- too easy to share data! It's really easy for threads to mess with each other's data and produce really confusing errors

Memory Models of Threads

A variable x is *shared* if and only if multiple threads reference some instance of x

Conceptually, threads behave like we discussed above:
- multiple threads run in the context of a single process
- they have their own thread context, stack, but share the rest of the process context

In practice, however, this model is not strictly enforced:
- Register values are truly separate and private, BUT
- Any thread can read and write the stack values of any other thread

Variable Types

Global Variable
- Variable declared outside of a function
- Virtual memory contains exactly one instance of any global variable

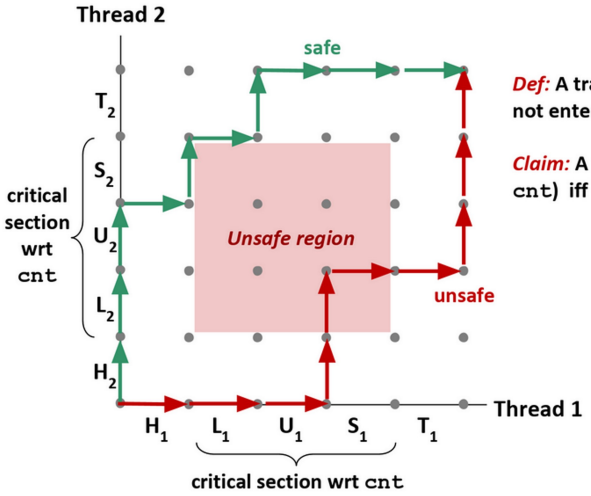
Local Variable
- Variable declared within the scope of a function, but without the *static* key
- Each thread stack contains its own copy of each local variable

Local Static Variable
- Variable declared locally using the *static* prefix
- Virtual memory contains exactly one instance of any local static variables

Concurrent Execution with Threads

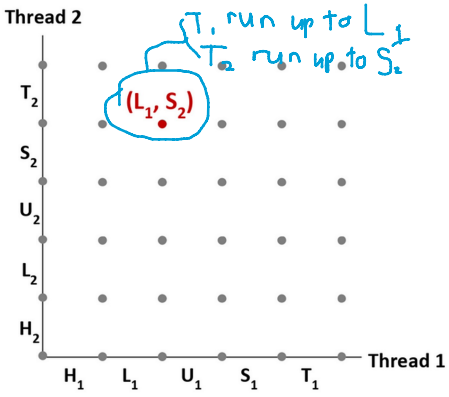
Theoretically possible to interleave the execution of any program containing threads. However, this is often problematic, due to shared variables messing with stuff.

We can track this funkiness by using progress graphs
X and Y axes correspond to the sequential order of instructions in a thread



Def: **A trajectory is *safe* iff it does not enter any unsafe region**

Claim: **A trajectory is correct (wrt cnt) iff it is safe**



To make sure that this always works, we're going to use Semaphores

Semaphore - a type of variable used to control access to shared data structures. Keeps track of available resources, and is coupled with methods to adjust this value safely (avoiding a race)

Invariant: Semaphore value >= 0 (never negative!)

given some semaphore S:
P(s) [aka sem_wait() for pthreads]
- if s is nonzero, decrement by one and return immediately
  - test/decrement operations are atomic
- If s is zero, suspend the thread until s becomes nonzero again and the thread is restarted by V(s)
- after restarting in ^ case, P(s) decrements s again and returns to caller

V(s) [aka sem_post() in pthreads]
- Increment s by 1
  - increment op is atomic
- If there are any threads waiting in a P operation for s to become 1, then restart exactly ONE of those threads

pthreads:

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

```
int sem_wait(sem_t *s); /* P(s) */

int sem_post(sem_t *s); /* V(s) */
```

add_item()
remove_item()

Mutual Exclusion:

Mutex - mutual exclusion using a binary semaphore (named mutex)

uses a binary semaphore - whose value is always 0 or 1

P operation "locks" the mutex
V op "unlocks"/"releases" the mutex

Contrasts with a Counting Semaphore - used as a counter of available resources

THIS WORKS - however, it is ORDERS OF MAGNITUDE slower than not using semaphores (but, you know, without them it doesn't work at all…)

A Process is DEADLOCKED if and only if it's waiting for a condition that will never be true

ABSTRACTION IS GOOD, BUT HAS LIMITS