ECE 531/598 – Advanced Operating Systems
Lecture 11

Vince Weaver

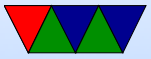
https://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

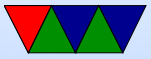
3 October 2023

Announcements

- 531 course has been officially approved!
- Note: Midterm after Fall break, on 17th
- Homework #5 Posted
- Some notes:
  - Review of C string handling, strcmp/strncmp and strcpy/strncpy/strlcpy
  - Be careful with the sizeof() operator, especially with strings. `sizeof(char[BUFSIZ])` vs `sizeof(char *)`
  - Talk about software engineering best practices. Unit

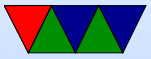


tests for printf. Code commenting. Source code versioning (git). We have been a bit lazy in this class.



HW#3 Review – Code

- Wasn't as picky this time, but comment code!
- Serial port: most got value right
  forgot to warn about floating point (say you wanted this to be parameterizable)
  Can you use floating point in kernel?
- printk: instead of /10, print remainder plus '0' instead /16 (which converts to shift) and two cases. 0-9 same as before, but A-F (you can just add 'A'-10 which I think is 55)



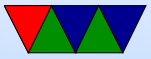
Beauty of ASCII. Often complicated use of ternary operator

technically upper vs lowercase %X vs %x

Can also use lookup table.

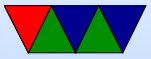
Be careful shifting, what if print 0xffffffff? Shift right? Be sure unsigned! Also in C, shift right by 32?

- Be sure print hardware info (r1)



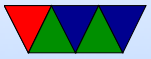
HW#3 Review – Questions

- Why serial port?
- What is parity? Why is it disabled?
  Faster (one fewer bit per byte), much bigger infrastructure to handle, not even that great (only detect one bit flip). Not that critical for text. What would a file transfer do? (checksum?)
  Some systems might not support? True, but why don't they support it?
- inline asm lets you write code that's not possible in C.



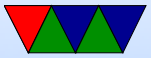
Also lets you bypass compiler (if you think you can do better)
Don't confuse it with the volatile keyword.

- Why no strtok()?
  strtok() given a string and set of delimiters (like space, tab) split up a line. So "led on" you'd get led, then run it again and get "on"
  string.h is the header, contains no code. Just describes the routine in the C library.
- People seem to mostly have usb-serial going, though occasional odd issues with MacOS



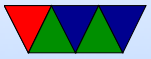
Processes

- What is a process?
- Your executable is loaded into memory and starts executing



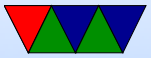
Process State

- Hardware state
  - registers (r0-r14), PC, status register
  - Floating Point / Vector? Performance Counters?
- Software/OS State
  - pid (process id), uid (user id)
  - Memory ranges, stack location, page tables
  - Process accounting / time stats
  - Open files (all open files, file offsets, etc)



Setting up the First Process

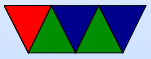
- We discussed how Linux starts userspace and starts process 1, traditionally init
  these days systemd. Can specify at kernel command line
- init becomes the parent of all userspace processes
- `fork()` is used to duplicate a process (make a copy of its process info, only difference being new process id)
- `exec()` is used to replace the current process with an executable from disk



Internally How do you Make a Process

create_process() or similar

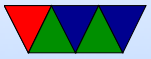
- Allocate memory for a new process structure
  array or maybe linked list
- Set the process ID
- Allocate a stack
- Initialize the registers
- Called by fork(), but with the extra step that it copies over the info from parent (including stack contents?)



Loading an Executable

load_process() or similar

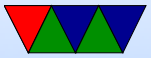
- Used by exec()
- Load executable from disk
- Parse the executable headers
- Allocate memory for machine code and data
- Allocate/zero the BSS memory
- Set up the program counter to point to entry point



Freeing a Process

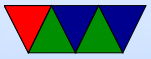
delete_process() or similar

- Used by exit() / exit_group()
- Close all open files
- Free all memory
- Possibly let parent know and wait until acknowledged
- Pass exit/return value back to parent
- Remove process from list

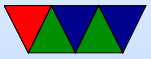


Kernel Process Creation

- Involves assembly language trickery
- Kernel can create threads
  - idle thread, pid 0, often does nothing (might halt/wfi to enter sleep mode when nothing else running)
  - Linux has a bunch of kernel worker threads it creates to help out with things, that look like processes but are parts of the kernel
- Kernel also does a bit of extra work to get init going because it has to be started in kernel space before

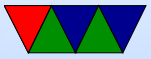


switching to user space



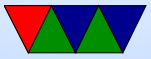
Kernel Process Creation

- First set up user registers. How do you do this from kernel/supervisor mode? Tricky, ARM created a special "system" mode (user+permissions) to make this easier.
- Set up stack
- Set the SPSR and link register to act as if we were returning from an exception, but with the return address the start of our user program.
- Return



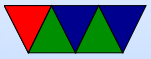
Multi-tasking / Multi-programming

- You could design a computer to only run one thing at a time
- Much more convenient if you can run multiple programs
  - If one program stops to wait for I/O, another can run
  - You can have multiple tasks going, but given the illusion they are all running at once
  - Especially useful when only have one core to run on



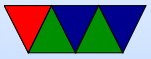
Timer Interrupt

- Usually a timer interrupt is set up to trigger every so often
- 250Hz on Linux
- If more than one process wants to run, the old one is stopped and its context is saved, and a new one is brought in to replace it



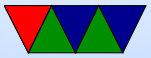
Context switching

- First time you get it working you get excited about having an AAA program and BBB programing printing ABABABA



Example Process Control Block

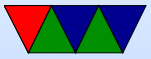
r14	the process LR
r13	
r12	
r11	
r10	
r9	
r8	
r7	
r6	
r5	
r4	
r3	
r2	
r1	
r0	PCB pointer points here (for stm instruction)
lr	pc from process to return to
spsr	



ARM Context Switch

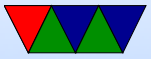
r12 = new process PCB, r13 = old

```
STM     sp,{R0-lr}^             ; Dump user registers above R13.
                ; ^ means get user register
MRS     R0, SPSR                ; get the saved user status
STMDB   sp, {R0, lr}            ; and dump with return address below.
                ; lr is the handler lr, pointing
                ; to pc we came from
LDR     sp, [R12], #4           ; Load next process info pointer.
CMP     sp, #0                  ; If it is zero, it is invalid
LDMDBNE sp, {R0, lr}            ; Pick up status and return address.
MSRNE   SPSR_cxsf, R0           ; Restore the status.
LDMNE   sp, {R0 - lr}^          ; Get the rest of the registers
NOP
SUBSNE pc, lr, #4               ; and return and restore CPSR.
                                ; Insert "no next process code" here.
```



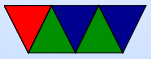
Storing

```
ldmfd   r13!,{r0-r3,r12,r14}
ldr r13,=PCB_PtrCurrentTask
ldr r13,[r13]
sub r13,r13,#offset15regs
stmia   r13,{r0-r14}^
mrs r0,spsr
stmdb   r13,{r0,r14}
```



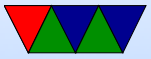
Loading

```
ldr r13,=PCB\_PtrNextTask
ldr r13,[r13]
sub r13,r13,#offset15regs
ldmdb   r13,{r0,r14}
msr spsr_cxsf,r0
ldmia   r13,{r0=r14}^   ; ^ means update user regs
ldr r13,=PCB_IRQstack
ldr r13,[r13]
movs    pc,r14
```



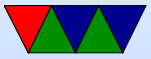
The Scheduler

- If you have multiple processes ready to run, how do you pick which to run next?
- The code that does this is called the scheduler
- This is a complex problem
- You want to run as fast as possible as it runs on every timer tick



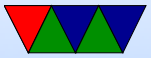
When to Schedule

- Task voluntarily yields (it has run out of work to do)
- If kernel blocks on I/O (Can be but to sleep instead of busy waiting)
- If timeslice runs out



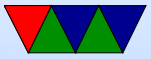
Scheduling Goals

- All: fairness, balance
- Batch: throughput (max jobs/hour), turnaround (time from submission to completion), CPU utilization (want it busy)
- Interactive: fast response, doesn't annoy users
- Real-time: meet deadlines, determinism



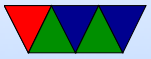
Batch Scheduling

- First-come-first-served (what if 2-day long job submitted first)
- Shortest job first
- Many others



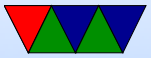
Interactive Scheduling

- Round-robin
- Priority – “nice” on UNIX
- Multiple Queues
- Others (shortest process, guaranteed, lottery)
- Fair scheduling – per user rather than per process



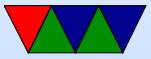
Real-time Scheduling

- Complex, more examples in 471 or real time OS course



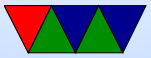
Scheduling Queues

- generally there will be a queue data structure holding all processes ready to run
- There will also be wait queues, where programs waiting on I/O can sleep
- If I/O comes in, the kernel will wake the process by moving it to the ready-to-run queue so it can be scheduled



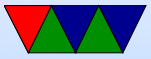
Process States

- Running – on CPU
- Ready – ready but no CPU available
- Blocked – waiting on I/O or resource
- Terminated – might stick around until parent acknowledges



The Linux Scheduler

- People often propose modifying the scheduler. That is tricky.
- Scheduler picks which jobs to run when.
- Optimal scheduler hard. What makes sense for a long-running HPC job doesn't necessarily make sense for an interactive GUI session. Also things like I/O (disk) get involved.
- You don't want it to have high latency

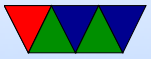


- Linux originally had a simple circular scheduler. Then for 2.4 through 2.6 had an O(N) scheduler

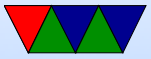
- Then in 2.6 until 2.6.23 had an O(1) scheduler (constant time, no many how many processes).

- Currently the "Completely Fair Scheduler" (with lots of drama). Is O(log N). Implementation of "weighted fair queuing"

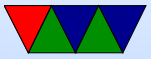
- How do you schedule? Power? Per-task (5 jobs, each get 20%). Per user? (5 users, each get 20%).



Per-process? Per-thread? Multi-processors? Hyper-threading? Heterogeneous cores? Thermal issues?

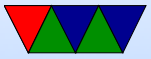


Linux Scheduler Details



Threads

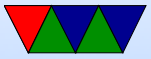
- Each process has one address space and single thread of control.
- It might be useful to have multiple threads share one address space
  GUI: interface thread and worker thread?
  Game: music thread, AI thread, display thread?
  Webserver: can handle incoming connections then pass serving to worker threads
  Why not just have one process that periodically switches?



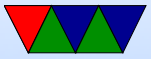
- Lightweight Process, multithreading

- Implementation:
  Each has its own PC
  Each has its own stack

- Why do it?
  shared variables, faster communication
  multiprocessors?
  mostly if does I/O that blocks, rest of threads can keep going
  allows overlapping compute and I/O

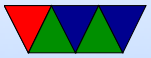


- Problems:
  What if both wait on same resource (both do a scanf from the keyboard?)
  On fork, do all threads get copied?
  What if thread closes file while another reading it?



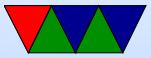
Common Thread Routines

- pthreads
  thread_init()
  thread_create() – specify function
  thread_exit()
  thread_yield() – if cooperative



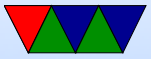
Thread Implementations

- Cause of many flamewars over the years



User-Level Threads (N:1 one process many threads)

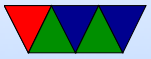
- Benefits
  - Kernel knows nothing about them. Can be implemented even if kernel has no support.
  - Each process has a thread table
  - When it sees it will block, it switches threads/PC in user space
  - Different from processes? When thread_yield() called it can switch without calling into the kernel (no slow



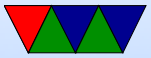
kernel context switch)
  - Can have own custom scheduling algorithm
  - Scale better, do not cause kernel structures to grow

- Downsides
  - How to handle blocking? Can wrap things, but not easy. Also can't wrap a pagefault.
  - Co-operative, threads won't stop unless voluntarily give up.

    Can request periodic signal, but too high a rate is inefficient.

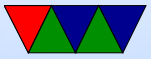


– Can't take advantage of multiple CPUs

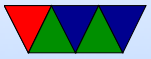


Kernel-Level Threads (1:1 process to thread)

- Benefits
  - Kernel tracks all threads in system
  - Handle blocking better
- Downsides
  - Thread control functions are syscalls
  - When yielding, might yield to another process rather than a thread

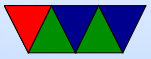


– Might be slower



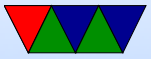
Hybrid (M:N)

- Can have kernel threads with user on top of it.
- Fast context switching, but can have odd problems like priority inversion.



Green Threads

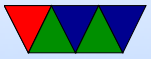
- Managed by virtual machine
- Java



Misc

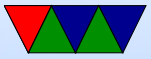
- Pop-up threads? Thread created for incoming message?

- adding multithreading to code?
  How to handle global variables (errno?)
  Thread-safe functions. Is strtok thread-safe? malloc? any routine that might not be re-entrant
  How are multiple stacks handled? One option each thread gets own copy of global variables. This can't be expressed by default in C, you need special routines, thread-local variables.

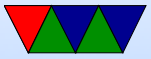


POSIX Threads (pthreads)

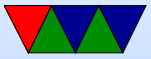
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS     10

void *perform_work(void *argument) {

    int value;

    value = *((int *) argument);
    printf("Thread␣with␣argument␣%d!\n", value);

    return NULL;
}

int main(int argc, char **argv) {

    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
```



```
int result, i;

/* create threads one by one */
for (i = 0; i < NUM_THREADS; i++) {
    thread_args[i]=i;
    printf("Main: creating thread %d\n", i);
    result = pthread_create(&threads[i],
        NULL, perform_work, (void *) &thread_args[i]);
    if (result!=0) {
        fprintf(stderr,"ERROR!\n");
        return -1;
    }
}

/* wait for each thread to complete */
for (i = 0; i < NUM_THREADS; i++) {
    /* block until each thread completes */
    result = pthread_join(threads[i], NULL);
    printf("MAIN: thread %d has completed\n", i);
    if (result!=0) {
        fprintf(stderr,"ERROR!\n");
        return -1;
    }
}
```

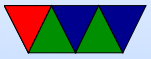


```
    printf("MAIN: All threads completed successfully\n");

    return 0;
}
```



POSIX Threads (pthreads) programming

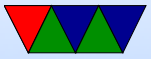
- Pass `-pthread` to gcc
- Thread management
  - `pthread_create (thread,attr,start_routine,arg)`
  - `pthread_exit (status)`
  - `pthread_cancel (thread)`
  - `pthread_attr_init (attr)`
  - `pthread_attr_destroy (attr)`
  - `pthread_join (threadid,status)` – blocks thread



until specified thread finishes

- `pthread_detach (threadid)`
- `pthread_attr_setdetachstate (attr,detachstate)`
- `pthread_attr_getdetachstate (attr,detachstate)`
- `pthread_attr_getstacksize (attr, stacksize)`
- `pthread_attr_setstacksize (attr, stacksize)`
- `pthread_attr_getstackaddr (attr, stackaddr)`
- `pthread_attr_setstackaddr (attr, stackaddr)`

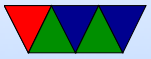
- Mutexes (synchronization)
  - `pthread_mutex_init (mutex,attr)`



- `pthread_mutex_destroy (mutex)`
- `pthread_mutexattr_init (attr)`
- `pthread_mutexattr_destroy (attr)`
- `pthread_mutex_lock (mutex)`
- `pthread_mutex_trylock (mutex)`
- `pthread_mutex_unlock (mutex)`

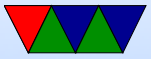
- Condition Variables – another way to synchronize

- Synchronization



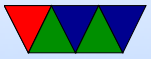
Linux

- Posix Threads
- Originally used only userspace implementations. GNU portable threads.
- LinuxThreads – use clone syscall, SIGUSR1 SIGUSR2 for communicating.
  Could not implement full POSIX threads, especially with signals. Replaced by NPTL
  Hard thread-local storage



Needed extra helper thread to handle signals
Problems, what happens if helper thread killed? Signals broken? 8192 thread limit? proc/top clutter up with processed, not clear they are subthreads

- NPTL – New POSIX Thread Library
  Kernel threads
  Clone. Add new futex system calls. Drepper and Molnar at RedHat
  Why kernel? Linux has very fast context switch compared to some OSes.
  Need new C library/ABI to handle location of thread-



local storage

On x86 the fs/gs segment used. Others need spare register.

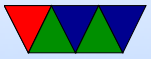
Signal handling in kernel

Clone handles setting TID (thread ID)

exit_group() syscall added that ends all threads in process, exit() just ends thread.

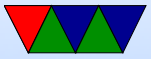
exec() kills all threads before execing

Only main thread gets entry in proc



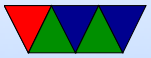
Misc

- adding multithreading to code?
  How to handle global variables (errno?)
  Thread-safe functions. Is strtok thread-safe? malloc? any routine that might not be re-entrant
  How are multiple stacks handled? One option each thread gets own copy of global variables. This can't be expressed by default in C, you need special routines, thread-local variables.

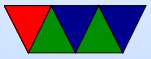


IPC – Inter-Process Communication

- Processes want to communicate with each other. Examples?
- Two issues:
  getting the message across
  synchronizing
- signals
- network, message passing (send, receive)



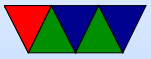
- shared memory (mmap)



Linux

- Signals and Signal handlers
  Very much like interrupts
  Concurrency issues much like threading

- Pipes
  stdout of one program to stdin of another
  one-way (half duplex)
  ls — sort
  pipe system call / dup



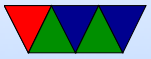
C library has popen()

- FIFOs (named pipes)
  exist as file on filesystem

- SystemV IPC
  shared memory, semaphores ipcs

- Just use mmap

- Unix domain sockets
  Can send file descriptors across



- Splice – move data from fd to pipe w/o a copy? VM magic?
- Sendfile. zero copy?