ECE 531 – Advanced Operating Systems
Lecture 16

Vince Weaver

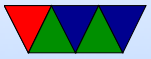
https://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

8 October 2025

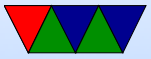
Announcements

- Note: Midterm after Fall break, on 20th
- Homework #5 was Posted
- Final Project description posted to website, more on that later



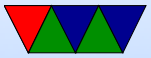
Scheduler Inputs

- How do you decide what to schedule?
- We mentioned you can just have queue with jobs ready to run
  How do you pick?
- Round-robin? Random?



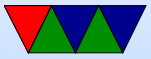
Fair Scheduling – Per User

- Evenly split between ready tasks: Per-task (5 jobs, each get 20%)
- What if one user created 96 jobs, 4 others 1 each?
  One user would get 96% of CPU and others 1%
- Fair scheduling would make it so still each user got fair share
  (5 users, each get 20%).

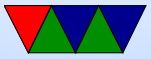


Hardware Concerns

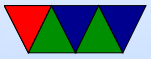
- What if multiple CPUs/Cores? How should they be assigned?
- What if 4 cores, 4 programs, just one each
  What if 5 programs? One is going to have to be swapped in/out, do you swap same core, do you spread the pain across all cores?
- Thread affinity
  - Ideally program stay on same core as much as possible
  - Things like Caches, Branch Predictors, other program



state build up on core and lost if a thread "migrates" to another core
  - NUMA issues too. Different cores closer to different parts of DRAM. If you migrate then RAM latency can go up unless you migrate RAM too
- IPC (inter-process communication)
  - If two threads communicating / working together you probably want them to both be active at the same time so they can work together
  - If not scheduled at same time adds latency to communications

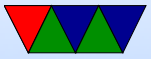


- Thermal Issues
  - Modern cores can “turbo boost” if low thermal load
  - This works best if cores being used far apart so the heat is spread more evenly across processor
  - Scheduler needs to be aware of this
- Low Power
  - If have multiple CPUs can maybe power one down if idle
  - If you can move processes all to one CPU then you can shut down the others
- Hyperthreading / SMT

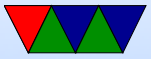


- Some processors have "hardware multithreading" where it looks like there are two cores but really it's not, it's reusing pipeline stages to allow multiple threads on one core
- Often running two threads can be slower than just running one thread on the core
- When scheduling, want to assign tasks to the hyperthread as last resort

- Heterogeneous CPUs
  - big/LITTLE, intel Power/Efficiency
  - how to decide which jobs run on power cores, which

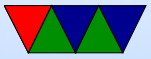


on efficiency



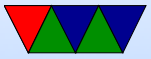
The Linux Scheduler – Challenges

- 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



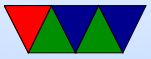
The Linux Scheduler – Challenges

- 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" (CFS) (with lots of drama). Is O(log N). Implementation of "weighted fair queuing"
- Would you want a O(N^3) scheduler?



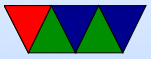
Linux Scheduler – Problems

- "The Linux Scheduler: A Decade of Wasted Cores" from Eurosys 2016 conference
- TODO: summarize points from it



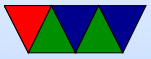
Linux Scheduler – EEVDF Scheduler

- Earliest Virtual Deadline First Scheduler
- `https://lwn.net/Articles/969062/`
- New scheduler, merged in Linux 6.6
- TODO: more notes on this



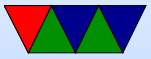
Single-Thread Processes

- A process is a program running on a computer, usually being managed by an operating system
- Process has one view of memory, one program counter, one set of registers, one stack

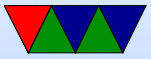


Could you build a multi-cpu program using just Processes?

- Yes. Start or fork() many copies, and have them communicate via message passing (this is more like a distributed system)
- Use Inter-Process Communication (IPC) Linux has many, all have tradeoffs
  - Sockets (UNIX domain, net)
  - Anonymous memory (mmap)
  - Files (on disk or mmaped)

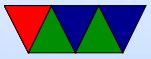


- Signals
- Pipes (Anonymous, Named, FIFOs)
- Shared Memory (SysV, POSIX)
- Message Queues (SysV, POSIX)
- Semaphores (SysV, POSIX)
- Futex locks
- Inotify
- FUSE
- D-Bus
- others?



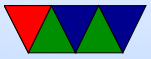
Threads

- Can an address space have multiple threads of control running in it at once?
- Examples when this might be useful:
  - GUI: interface thread and worker thread?
  - Game: music thread, AI thread, display thread?
  - Webserver: can handle incoming connections then pass serving to worker threads



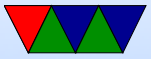
Multithreading Implementation

- The memory layout is shared by all threads in a process
- Each thread has its own PC
- Each thread has its own stack
- Each thread has its own copy of the register file
- Each thread has its own "thread local storage" (TLS) area for private variables



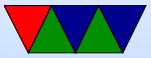
Multithreading Tradeoffs

- Benefits
  - shared variables, faster communication
  - If program blocks on I/O, rest of threads can keep going
  - programs can run faster on multiprocessor systems
- Complications
  - Resource conflicts: (what if multiple threads try to scanf() at same time?)
  - What if a thread closes a file while another is trying



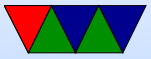
to read?

- On a fork, does the child process have the same threads or not?



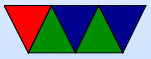
Thread Implementations

- Cause of many flamewars over the years



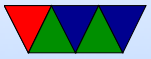
User-Level Threads (N:1 one process many threads)

- 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



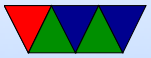
kernel context switch)
  - Can have own custom scheduling algorithm
  - Scale better, do not cause kernel structures to grow
- Downsides
  - How to handle blocking? Can wrap things, but not easy. Also can't wrap a pagefault.
  - Co-operative, threads won't stop unless voluntarily give up.

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



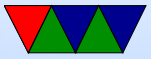
Kernel-Level Threads (1:1 process to thread)

- 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



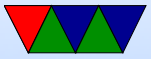
Hybrid (M:N)

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



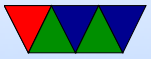
Common Thread Programming Models

- Pipeline – task broken into a set of subtasks that each execute serial on own thread
- Manager/worker – a manager thread assigns work to a set of worker threads. Also manager usually handles I/O static worker pool – constant number of threads dynamic worker pool – threads started and stopped as needed
- Peer – like manager/worker but the manager also does calculations



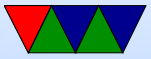
Shared Memory Model

- All threads have access to shared memory
- Threads also have private data
- Programmers must properly protect shared data



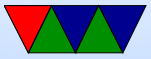
Thread Safety

- Is a function called thread safe?
- Can the code be executed multiple times simultaneously?
- The main problem is if there is global state that must be remembered between calls. For example, the `strtok()` function.
- As long as functions only use local variables (on stack, not static or global) usually not an issue.
- Some issues can be addressed with locking.



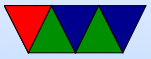
POSIX Threads (pthreads)

- Standardized thread interface
- Standard cross-platform set of routines to use



Other types of Threads

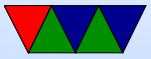
- co-routines
  - Not found in C
  - Sort of like co-operatively scheduled software threads
  - functions can be suspended at various points and re-started later
  - Less issues than full threads as only one can run at a time, simplifying locking
- C11/C17 threads
  - `https://beej.us/guide/bgc/html/split/multithr`



`html`

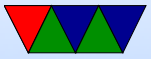
- Sort of last-minute addition to C
- No real benefits over using pthreads?

- Fibers?
- Green Threads

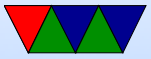


Linux Threading – Historical

- Linux original thread implementation was horrible software based
- Originally used only userspace implementations. GNU portable threads.
- LinuxThreads (no longer used) – use clone syscall, SIGUSR1 SIGUSR2 for communicating.
  Could not implement full POSIX threads, especially with signals. Hard thread-local storage
  Needed extra helper thread to handle signals

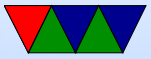


Problems, what happens if helper thread killed? Signals broken? 8192 thread limit? proc/top clutter up with processes, not clear they are subthreads



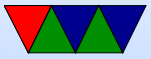
Linux Threading – NPTL

- NPTL – Native POSIX Thread Library
- Kernel threads
- Clone syscall, new futex system calls.
- Developed around 2003 or so by Drepper and Molnar at RedHat, Kernel 2.6
- 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



Brief Linux Pthread Programming Notes

- Pass `-pthread` to gcc
- pthread_create(function) – start new thread of execution, launching with function
- pthread_exit() – exit thread
- pthread_join() – wait until thread finishes
- lots and lots more