

ECE 598 – Advanced Operating Systems Lecture 23

Vince Weaver

`http://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

21 April 2016

Announcements

- Don't forget HW#9
- Midterm next Thursday



Process States

- Running – on CPU
- Ready – ready but no CPU available
- Blocked – waiting on I/O or resource
- Terminated



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 matter 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?



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?



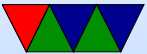
Common Thread Routines

- 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];
```



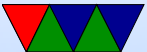
```

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.

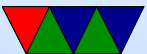
Signal handling in kernel

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.

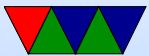


IPC – Inter-Process Communication

- 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?

