Synchronization and Spthreads

CIS 4480/5580 Recitation 6
October 29, 2025

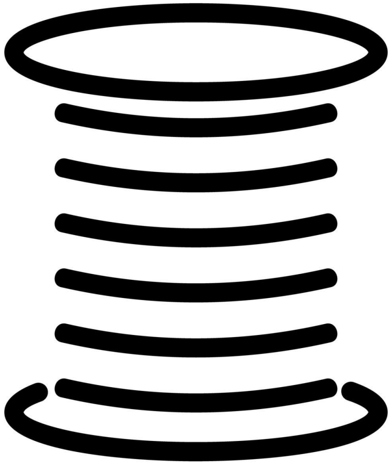
Try out Problem 1 First!

We'll go over it together in a bit!

Spthreads

Threads

- Single sequential execution path in a process
- Shares an address space and some resources with other threads grouped under the same process
    - This makes them more lightweight compar
      to forking a new process
- The OS kernel uses threads as schedulable unit
- In POSIX systems, you can use the pthread libr
  to manage threads



What are Spthreads?

Modified version of pthreads that where we can explicitly pause and resume execution

Synchronization

Synchronization is the coordinating of multiple execution streams to ensure they work together (to achieve some goal)

It guarantees the correctness of our work in a concurrent setting

Spthread Features

```
int spthread_create(
    spthread_t* thread,
    const pthread_attr_t* attr,
    void* (*start_routine)(void*),
    void* arg
);

int spthread_cancel(spthread_t thread);

void spthread_exit(void* status);

int spthread_join(
    spthread_t thread,
    void** retval
);

bool spthread_equal(
    spthread_t first,
    spthread_t second
);

bool spthread_self(spthread_t* thread);
```

Create, Cancel, Exit, Join, Equal, and Self

Pthread equivalents for spthread

```
int spthread_continue(spthread_t thread);

int spthread_suspend(spthread_t thread);

int spthread_suspend_self();
```

Continue, Suspend, Suspend Self

New functions to start and stop spthreads on demand

```
int spthread_disable_interrupts_self();

int spthread_enable_interrupts_self();
```

Disable and Enable Interrupts

Can help us avoid corrupted state (how?)

A Simple Scheduler

- For Milestone 0, we've provided sched-demo.c as an example scheduler
- Apart from Milestone questions, please note:
  - How an spthread is "cancelled"
  - How we control access to global variables
- When spthreads are created, are they running from the start?

schedular()

```c
static void scheduler(void) {
  int curr_thread_num = 0;

  // mask for while scheduler is waiting for
  // alarm to go off
  sigset_t suspend_set;
  sigfillset(&suspend_set);
  sigdelset(&suspend_set, SIGALRM);

  // just to make sure that
  // sigalrm doesn't terminate the process
  struct sigaction act = (struct sigaction){
      .sa_handler = alarm_handler,
      .sa_mask = suspend_set,
      .sa_flags = SA_RESTART,
  };
  sigaction(SIGALRM, &act, NULL);

  // make sure SIGALRM is unblocked
  sigset_t alarm_set;
  sigemptyset(&alarm_set);
  sigaddset(&alarm_set, SIGALRM);
  pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);

  struct itimerval it;
  it.it_interval = (struct timeval){.tv_usec = centisecond * 10};
  it.it_value = it.it_interval;
  setitimer(ITIMER_REAL, &it, NULL);

  // locks to check the global value done
  while (!done) {
    curr_thread_num = (curr_thread_num + 1) % NUM_THREADS;
    spthread_t curr_thread = threads[curr_thread_num];

    spthread_continue(curr_thread);
    sigsuspend(&suspend_set);
    spthread_suspend(curr_thread);
    // lock
  }
}
```

- What is pthread_sigmask vs sigprocmask?
- What does setitimer do?
- Why can this thread call sigsuspend?
  - How does this affect other threads?
- sp_cont and cp_sus don't return until the threads have truly been continued or suspended
  - It is akin to an acknowledgement using pthread signals.

Ostrich Algorithm

- Deadlock detection and fixes can be expensive
- In some contexts, it might be worth it to not plan for deadlocks
    - We can naively restart processes if they hang, for example
- The Linux kernel has no in-built deadlock detection/resolution mechanism
    - If your code deadlocks, then it will stay blocked until you restart some processes, for example

Really, this covers anything that we might think is highly unlikely and not worth fixing ***until it costs us more money not to fix it…***

Try Out Problem 2!

Synchronization Strategies

Atomics and Memory Ordering

Hardware and compiler level guarantees on the behavior of reads and writes

Backbone of other methods

Synchronization Primitives

Explicitly control access to resources

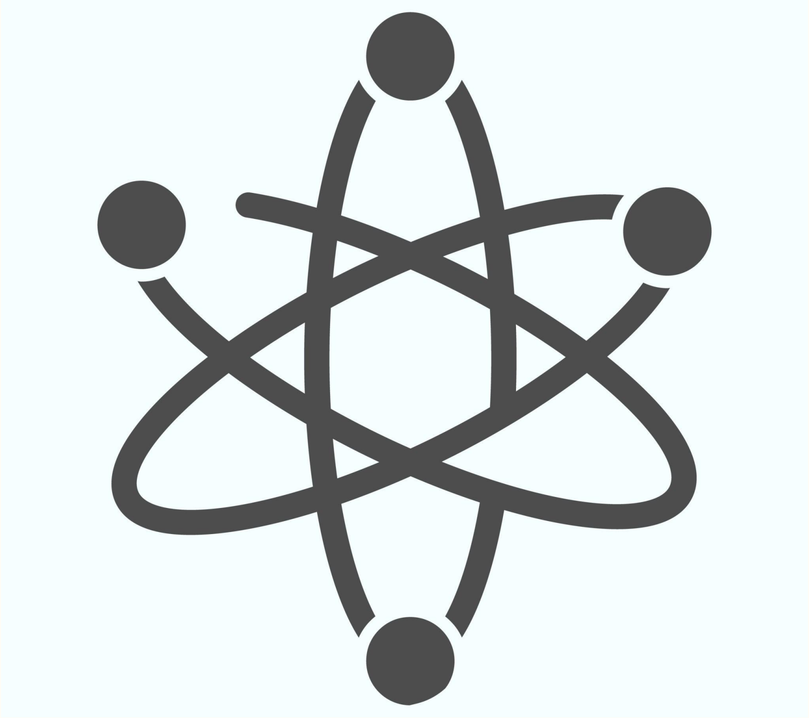
Examples include mutexes, semaphores, condition variables

Lock-free Programming

Uses atomics and memory ordering to bypass explicit synchronization primitives and reduce overhead in multicore settings

Atomicity

- Refers to operations that are not interruptible by some other specified operations (depends on the primitive)
- Guarantee provided by collaboration of hardware, compiler, or the OS/libraries (POSIX)
- Many modern instruction sets offer atomic modifiers (LOCK in x86_64)
- Compiler ensures our declarations in our (C) code are respected in assembly
- OS/libraries provide some higher level abstractions of primitives that we can use (mutex)



Memory Ordering

- Memory accesses (reads and writes) don't always happen in the way we explicitly write them in code
- Out-of-order execution, compiler optimizations, etc.
- Can enforce a deterministic order with assembly instructions like **fences** and ordering declarations in our code
- **Not a big concern in this class**

volatile sig_atomic_t

volatile: compilers don't optimize memory accesses via ideas like register storage or reordering (important with mmap I/O side effects) and we can safely assign values in signal handlers

sig_atomic_t: (only) guarantees accesses occur without interruption by signals

Their effects stack, so we use them together in values modified by handlers (PID tracking in penn-shell)

But we still need to gate access to this variable across parallel threads! C11 introduced _Atomic for truly atomic writes and reads across threads. `<stdatomic.h>`

```
atomic int x; //give you an atomic int! Extends to other types…
```

Learn more by checking out the gnu pages.

Synchronization Primitives

Mutexes

Only one thread can own a mutex lock at any point in time

Great for gating access to a shared resource

Condition Variables (Today's Lecture)

Threads wait for a condition (shared variable locked by mutex) to become true before continuing

Semaphores (Today's Lecture)

Counter-based primitive, where there is a total capacity and each acquisition decreases capacity atomically (and releases increase it)

Linux uses a file-like interface for semaphores (why?)

2 Lock Types

Test-Set Lock

Note: the bool starts off as False

Atomically read the last boolean state value and set it to True.

If the last read value was True, then we were not the first, so we don’t get the lock and keep trying to acquire

Otherwise, we now have the lock

Thanks! Any questions?

Race Conditions

When behavior depends on ordering of concurrent operations

Remember setpgid race condition in shell

We can also have race conditions in accesses to shared memory (2+ threads modifying a shared value without any synchronization)

Synchronization primitives and atomics can help us mitigate race conditions

Deadlock

New problem with locks, potentially we can halt our program

If a worker is waiting on a resource that will never be released, that worker can't continue

Antithesis of **liveness**

Formal requirements: mutual exclusion, hold and wait, no preemption, circular wait

Fixing Deadlocks

Removing Mutual Exclusion

Lock-free programming

Single threaded async approach

Atomic operations at hardware level

Avoiding Hold and Wait

Have thread acquire all required resources at once

Preemption

Force a thread to release a resource if no progress is made (maybe with a timeout)

Can be tough to recover to a checkpoint/snapshot

Avoiding Circular Wait

Resource acquisition shouldn't have circular dependencies

Enforcing an ordering in which we acquire resources across threads can help

Deadlock Detection

Resource Acquisition Graph:

- Each thread and resource is a node
- If a thread owns a resource, the resource node has an edge to the thread node
- If a thread wants a resource, the thread points to the resource node

Wait-For Graph:

- Each thread is a node
- If thread 1 waits on a resource held by another thread 2, there is a directed edge from thread 1 to thread 2

Cycles represent deadlocks

Other Issues

- Livelock
- Starvation
- Priority Inversion
- Lock Contention