

# THREADS AND SYNCHRONIZATION

Introduction to  
Systems Software

wilkie

# THREADS

Strings? Threads?? What are we building... a loom???

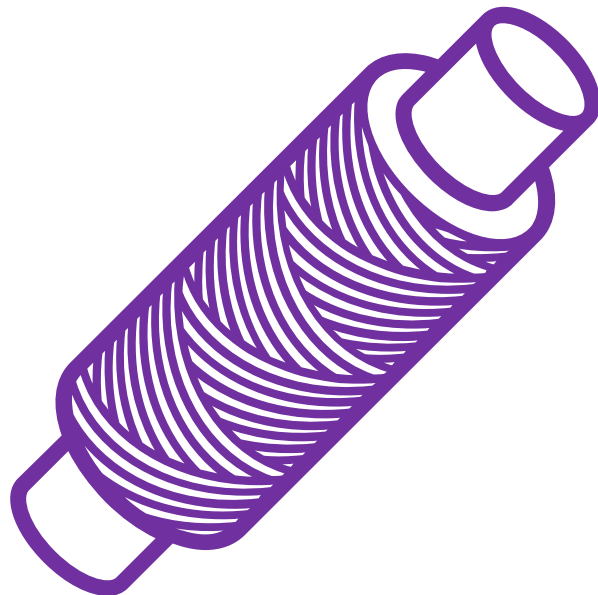
# Our story so far...

- We looked at how processes reproduce with `fork()`
  - This gave us some type of concurrency.
  - It is process-level, so the OS is scheduling each task.
- We saw some issues with concurrent programming.
  - Race conditions mean we have to much more carefully consider our code.
- This time...
  - We will look at other forms of concurrency.
  - Some new methods of coordinating the different sub-programs.
  - And some new... dreaded... types of concurrency bugs.



# Threads

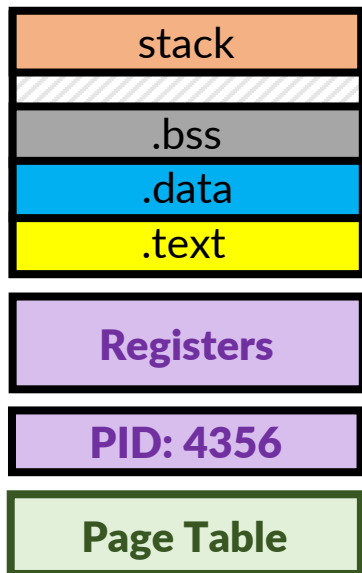
- Process-level concurrency with `fork()` is powerful, but inflexible.
  - The OS schedules the task, incurring context switching overhead.
  - The process memory is copied making it hard to share data among tasks.
- A **thread** is a concurrency primitive that is inner-process.
  - The program itself schedules the task as part of the same process.
  - Process memory, therefore, is shared across all threads.



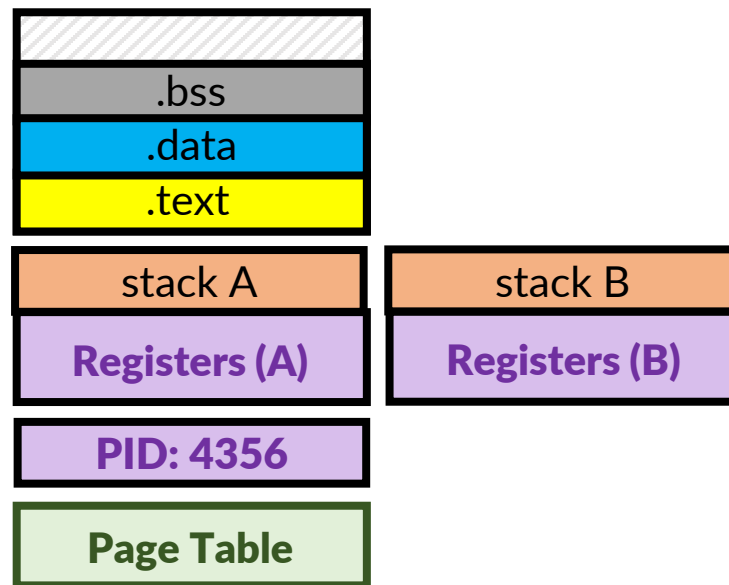
# Preparing the process for threads.

- In processes, we saw our traditional memory layout.

*Traditional  
Process*



*Threaded  
Process*



# libpthread

- The 2011 amendment to the C standard (C11) added a threading API.
  - However, we will still be looking at an older, more prevalent standard.
- We will be reviewing the pthread standard.
  - The C11 `threads.h` API is still very similar.
  - There are ports of the `pthread.h` interface to many OSes.
  - Lots of threading APIs in other language emulate it.
- Still very useful to learn!

# POSIX

- The “p” in pthread stands for the Portable Operating System Interface (**POSIX**).
  - This is a standard for creating OS abstractions.
  - Intended to lower the burden of porting applications.
    - Most OSes conform to most POSIX standards.
    - However, very few OSes fully implement POSIX.
- POSIX standardizes threads, but also process creation and the behavior of fork, file abstractions, and how data is shared among processes.
  - Many OS interfaces are POSIX interfaces and remain (mostly) true across different platforms.



# Creating... hmm... no... *weaving* a thread

```
C(gcc -o thrd_bad thrd_bad.c -lpthread)
```

```
#include <pthread.h>
```

```
#include <stdio.h>
```

```
void* thread(void* data) {  
    printf("Hello, %s!\n", data);  
    return NULL;  
}
```

← **This function runs in a thread**

```
void main(void) {  
    char* str = "wilkie";  
  
    pthread_t tid;  
    pthread_create(&tid, NULL, thread, (void*)str);  
    printf("Done!\n");  
}
```

← **Holds the thread ID.**

**The thread function** ← **Function argument**

- Here is a basic threaded program.
  - main() is within the main thread.

- The pthread\_create function creates a second thread, which runs alongside the main thread.
  - The first argument is the address of a variable to hold the thread ID.
  - The NULL is where you can add some flags, but the defaults are OK.
  - The thread is the function to use.
  - The last argument is passed to that function and generally an address.





# The race to the finish.

```
C(gcc -o thrd_bad thrd_bad.c -lpthread)
```

```
#include <pthread.h>
```

```
#include <stdio.h>
```

```
void* thread(void* data) {  
    printf("Hello, %s!\n", data);  
    return NULL;  This never happens!  
}
```

```
void main(void) {  
    char* str = "wilkie";  
  
    pthread_t tid;  
    pthread_create(&tid, NULL, thread, (void*)str);  
  
    printf("Done!\n");  
} 
```

**In the end of the program, threads are also all exited, potentially prematurely.**

```
> ./thrd_bad
```

```
Done!
```

- However, when the process exits normally, all threads are also canceled, even if they haven't completed.
- In this run, the second thread never prints its message.

# Being considerate

```
C(gcc -o thrd thrd.c -lpthread)
```

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

```
void* thread(void* data) {
    printf("Hello, %s!\n", (char*)data);
    return NULL;
}
```

```
void main(void) {
    char* str = "wilkie";

    pthread_t tid;
    pthread_create(&tid, NULL, thread, (void*)str);
```

```
    pthread_join(tid, NULL); // wait for thread
```

```
    printf("Done!\n");
}
```

← The "str" argument.

← Waits...

← Guaranteed to happen only after thread() completes.

```
> ./thrd
Hello, wilkie!
Done!
```

- pthread\_join() waits for the given thread to exit by thread ID.
  - The NULL is, again, optional flags.
- Here, the main thread waits until the thread function completes.
  - It prints out the string given by the argument.

# Sharing is caring

```
C(gcc -o thrd_share_bad thrd_share_bad.c -lpthread)
```

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

```
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        printf("THREAD: %d\n", counter);
        counter++;
    }
    return NULL;
}
```

← **Thread function increments**

```
void main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);

    while(counter < 100) {
        printf("MAIN: %d\n", counter);
        counter++;
    }
    printf("Done!\n");
}
```

← **Main thread increments, too!**

- Unlike process-level concurrency using `fork()`, threads share memory.
- Each thread, here, shares access to the same global variable `counter`.
  - When the main thread updates, the secondary thread sees that value.
- Threads share the same virtual address space (and page table.)
  - They only have their own stack and CPU state.

# A problem returns with a vengeance

```
C(gcc -o thrd_share_bad thrd_share_bad.c -lpthread)
```

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

```
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        printf("THREAD: %d\n", counter);
        counter++;
    }
    return NULL;
}
```

← **Thread function might get interrupted before the print**

```
void main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
```

```
    while(counter < 100) {
        printf("MAIN: %d\n", counter);
        counter++;
    }
    printf("Done!\n");
}
```

← **Then main thread increments! :(**

```
> ./thrd_share_bad
```

```
MAIN: 0
```

```
MAIN: 1
```

```
MAIN: 2
```

```
MAIN: 3
```

```
THREAD: 4
```

```
THREAD: 5
```

```
THREAD: 6
```

```
MAIN: 4
```

```
MAIN: 8
```

```
MAIN: 9
```

```
MAIN: 10
```

```
THREAD: 7
```

```
THREAD: 11
```

```
THREAD: 12
```

# What happened???

```
C(gcc -o thrd_share_bad thrd_share_bad.c -lpthread)
```

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

```
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        printf("THREAD: %d\n", counter);
        counter++;
    }
    return NULL;
}
```

← **Thread function increments**

```
void main(void) {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);

    while(counter < 100) {
        printf("MAIN: %d\n", counter);
        counter++;
    }
    printf("Done!\n");
}
```

← **Main thread increments, too!**

- Since the threads share memory, access to a variable, such as this, may require extra care.
- When the main thread gets interrupted just as it was printing the value, the thread is scheduled.
  - The thread prints the value instead.
  - Then the main thread, when it continues, prints it again!
- If only we had a way to... align them in time... what's the word...

# SYNCHRONIZATION

Stop! Hammer time!

# A story about the railroad

- Systems scientists have long been inspired by the real-world for insight on design.
- The rail system requires a lot of attention to detail to provide:
  - Orderly and timely scheduling of trains.
  - Shared use of a single resource: rail.
  - Coördination with trains and competing interests.
- In order to provide this, trains make use of signals and switching areas.
  - Trains wait while others pass, all agreeing on the nature of signals.
  - The signals are called *semaphores*.



Photo by David Ingham

# The seminal semaphore

- A **semaphore** is a special counter used for synchronization.
  - Invented by Dutch systems scientist Edsger Dijkstra in the early 1960s.
- The counter is a signed integer that often starts at zero or one.
- Two defined operations:
  - Up (signal/release); increments counter.
  - Down (wait/acquire); decrements counter but waits if the counter is 0.
- These operations often have different names or are abbreviated:
  - V (Based on Dutch *vrijgave* “to release”)
  - P (Based on Dutch *passering* “to pass”, *based around railroad terminology*)




# Semaphores to prevent the derailing

```
C(gcc -o thrd_share thrd_share.c -lpthread)
```

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
```

```
sem_t lock;
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        sem_wait(&lock); // down
        printf("THREAD: %d\n", counter);
        counter++;
        sem_post(&lock); // up
    }
    return NULL;
}
```

 **Critical Section**

```
void main(void) {
    sem_init(&lock, 0, 1); // create a counter starting at 1
```

```
pthread_t tid;
pthread_create(&tid, NULL, thread, NULL);
```

```
while(counter < 100) {
    sem_wait(&lock); // down
    printf("MAIN: %d\n", counter);
    counter++;
    sem_post(&lock); // up
}
printf("Done!\n");
}
```

- `sem_init()` creates a new semaphore.
  - The first argument is an address to a variable that will hold the semaphore data.
  - The second argument, when 0, means that other threads can see the semaphore. Non-zero means other threads cannot interact with the semaphore, which is a bit more advanced.
  - The third argument is the initial value.
    - Here it is 1.
- `sem_wait()` decrements the counter.
  - Waits to decrement if the counter is 0.
- `sem_post()` increments the counter.
  - May release a thread waiting at `sem_wait`

# Semaphores to prevent the derailing

```
C(gcc -o thrd_share thrd_share.c -lpthread)
```

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
```

```
sem_t lock;
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        ➡ sem_wait(&lock); // down
        printf("THREAD: %d\n", counter);
        counter++;
        sem_post(&lock); // up
    }
    return NULL;
}
```

↖ **Critical Section**

```
void main(void) {
    sem_init(&lock, 0, 1); // create a counter starting at 1

    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);

    while(counter < 100) {
        ➡ sem_wait(&lock); // down
        printf("MAIN: %d\n", counter);
        counter++;
        sem_post(&lock); // up
    }
    printf("Done!\n");
}
```

- When both threads hit `sem_wait()` at the same time, only one continues.
- When one sets the lock; other waits.
  - The other thread relies on the first to eventually release the lock using `sem_post()`
  - When this happens, the other thread can go.
- The lock/unlock pattern creates a **critical section**, a piece of code that has the guarantee that only one task can enter at a time.
  - Here, the counter is guaranteed to update at the same time as it is printed.

# Semaphores to prevent the derailing

**C**(gcc -o thrd\_share thrd\_share.c -lpthread)

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
```

```
sem_t lock;
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        sem_wait(&lock); // down
        printf("THREAD: %d\n", counter);
        counter++;
        sem_post(&lock); // up
    }
    return NULL;
}
```

← **Critical Section**

```
void main(void) {
    sem_init(&lock, 0, 1); // create a counter starting at 1

    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);

    while(counter < 100) {
        sem_wait(&lock); // down
        printf("MAIN: %d\n", counter);
        counter++;
        sem_post(&lock); // up
    }
    printf("Done!\n");
}
```

```
> ./thrd_share
```

```
MAIN: 0
```

```
MAIN: 1
```

```
MAIN: 2
```

```
MAIN: 3
```

```
THREAD: 4
```

```
THREAD: 5
```

```
THREAD: 6
```

```
MAIN: 7
```

```
MAIN: 8
```

```
MAIN: 9
```

```
MAIN: 10
```

```
THREAD: 11
```

```
THREAD: 12
```

```
THREAD: 13
```

# Mutex... ew... don't like the sound of that

- As you can see, there is a common case.
  - Simple critical sections just need a counter that covers 0 and 1.
- A **mutex** is a special Boolean used for synchronization.
  - It is short for “mutual exclusion,” a term for when two things can only have one resource at a time.
- There are two defined operations:
  - lock / wait; only precedes if the mutex is unlock.
  - unlock / release; unlocks the mutex.
- A mutex can be created using a semaphore.
  - It provides a subset of the capabilities of the more general semaphore.

# A mutex to prevent the derailing

**C** (gcc -o thrd\_share\_mutex thrd\_share\_mutex.c -lpthread)

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

```
pthread_mutex_t lock;
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        pthread_mutex_lock(&lock);
        printf("THREAD: %d\n", counter);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

← **Critical Section**

```
void main(void) {
    pthread_mutex_init(&lock, NULL);

    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);

    while(counter < 100) {
        pthread_mutex_lock(&lock);
        printf("MAIN: %d\n", counter);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    printf("Done!\n");
}
```

- Mutexes are useful for locking single resources.
  - It follows much the same pattern as semaphores, and perhaps easier to understand.
- pthread\_mutex\_init() creates the mutex similarly to sem\_init().
- pthread\_mutex\_lock() and pthread\_mutex\_unlock() do the locking and unlocking, as expected.

# A mutex to prevent the derailing

**C** (gcc -o thrd\_share\_mutex thrd\_share\_mutex.c -lpthread)

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

```
pthread_mutex_t lock;
int counter = 0;
```

```
void* thread(void* data) {
    while(counter < 100) {
        pthread_mutex_lock(&lock);
        printf("THREAD: %d\n", counter);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

← **Critical Section**

```
void main(void) {
    pthread_mutex_init(&lock, NULL);

    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);

    while(counter < 100) {
        pthread_mutex_lock(&lock);
        printf("MAIN: %d\n", counter);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    printf("Done!\n");
}
```

```
> ./thrd_share_mutex
```

```
MAIN: 0
```

```
MAIN: 1
```

```
MAIN: 2
```

```
MAIN: 3
```

```
MAIN: 4
```

```
THREAD: 5
```

```
THREAD: 6
```

```
THREAD: 7
```

```
THREAD: 8
```

```
THREAD: 9
```

```
MAIN: 10
```

```
MAIN: 11
```

```
THREAD: 12
```

```
THREAD: 13
```

# Strategies

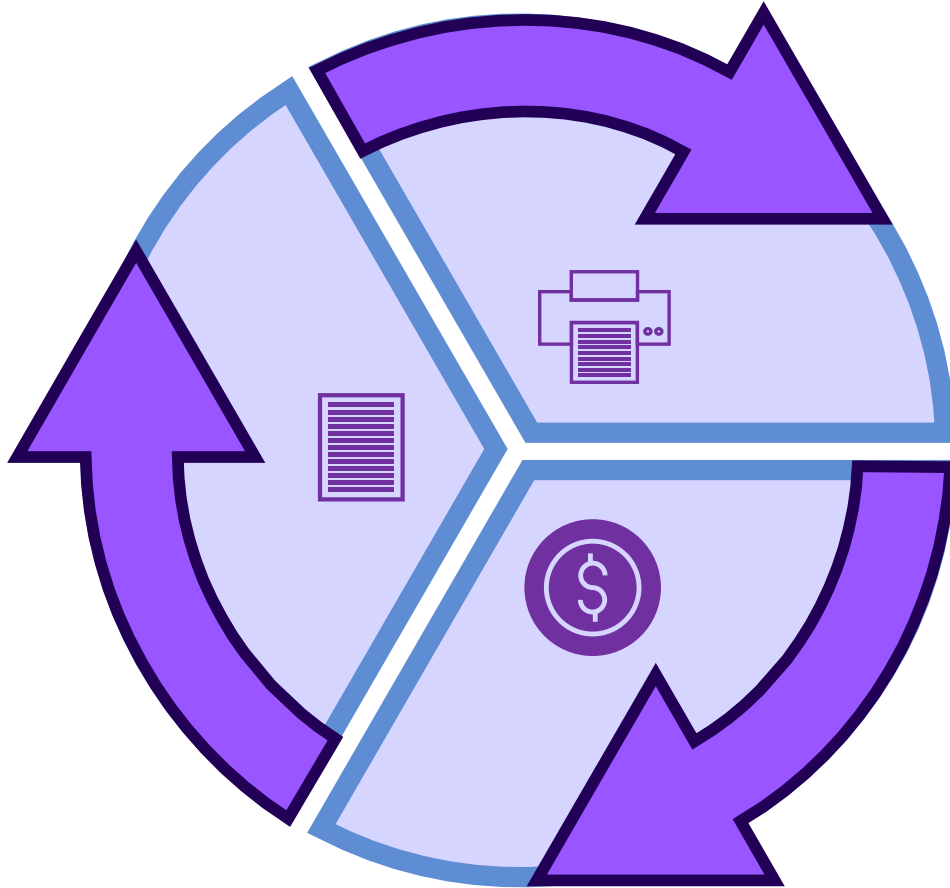
- Semaphores and mutexes are both primitives to aid in concurrent programming.
- We saw, here, another example of a **race condition**, a concurrency bug where the absence of guaranteed order can result in incorrect behavior.
  - Namely, threads being interrupted in-between operations that need to happen together and racing another thread that will incorrectly use that intermediate value.
- However, that's not the only type of concurrency bug we can have!
  - Yay! ☹️

# PARALLEL PITFALLS

This is like that time when a bird pooped on me the same time I stepped in a very muddy puddle.



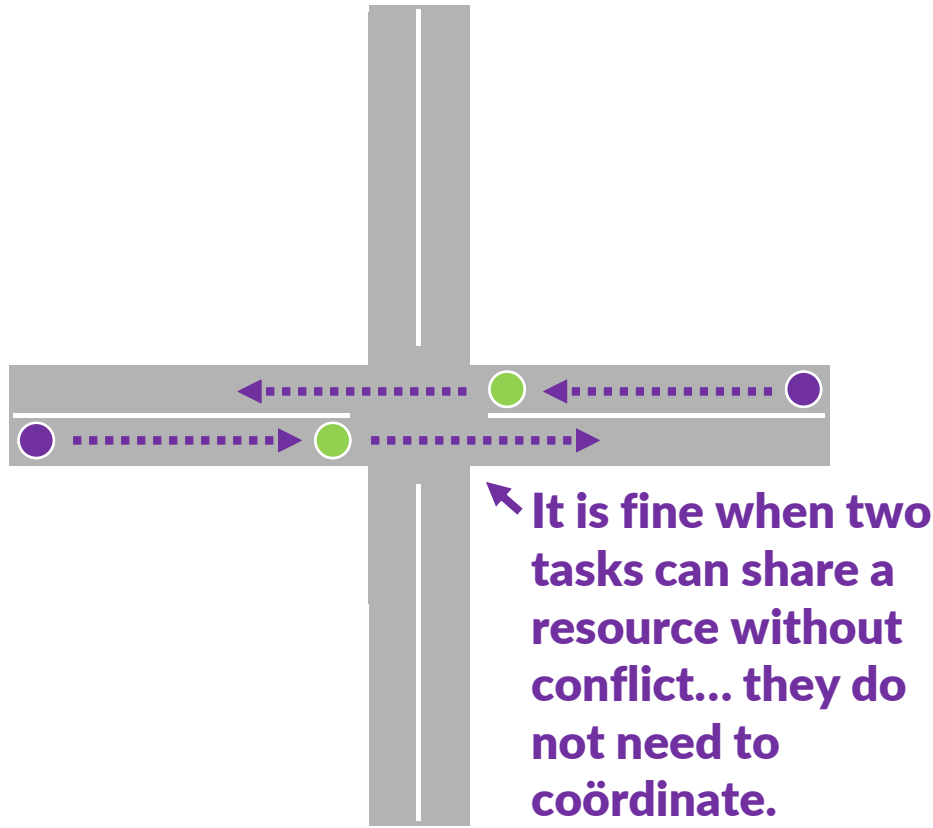
# Everybody loves resources



Resource contention:

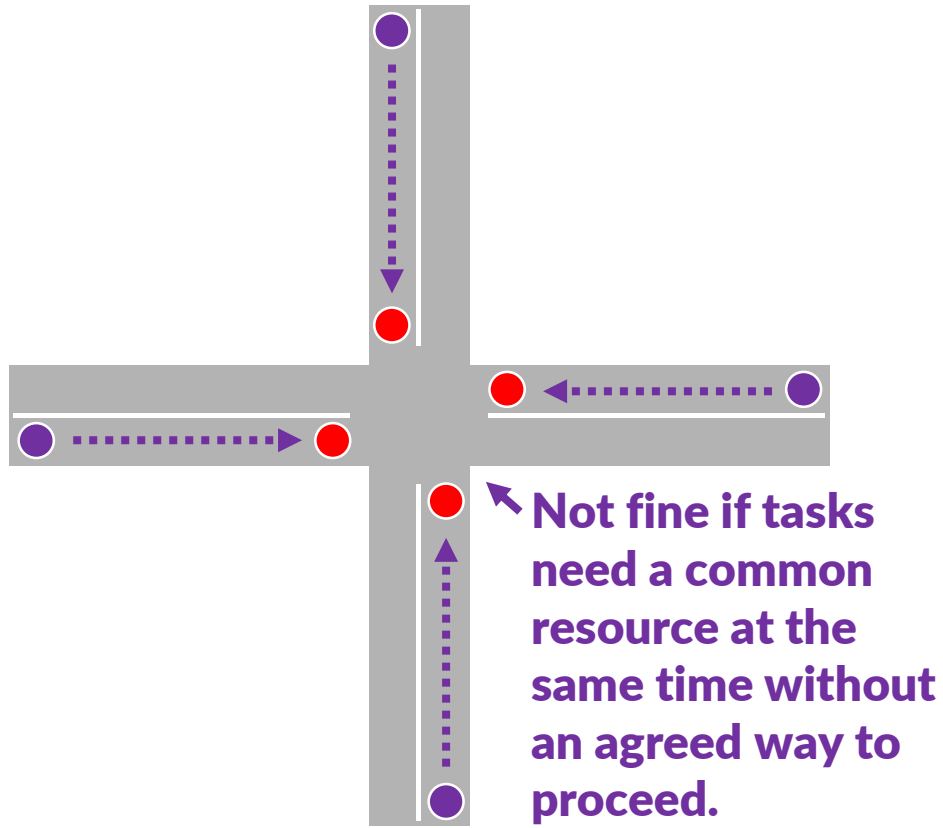
- Printer needs paper...
- You need to buy some paper...
- You need to print an order form for paper...
- Printer needs paper...

# Typically...



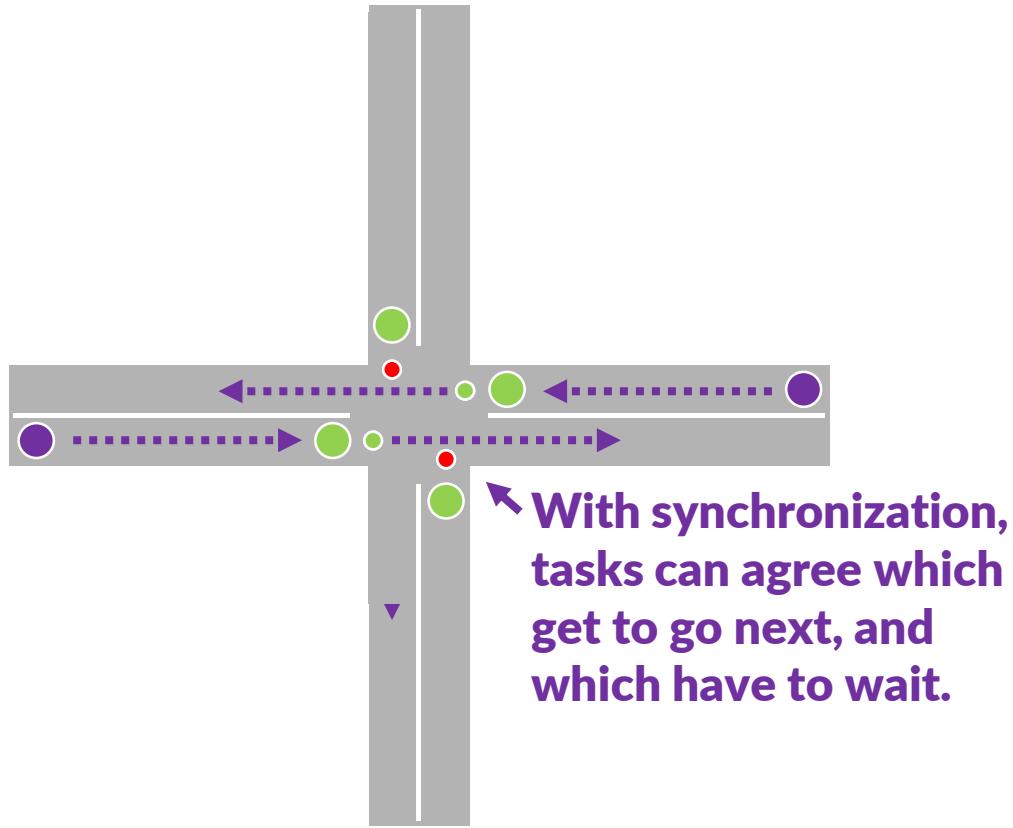
- Metaphor: intersection.
- The intersection is a shared resource, much like a device or the CPU.
- Multiplexing the intersection is important to avoid crashes.
- When the streets aren't busy, cars just make it safely across.

# Deadlock: The traffic jam (it's not very delicious)



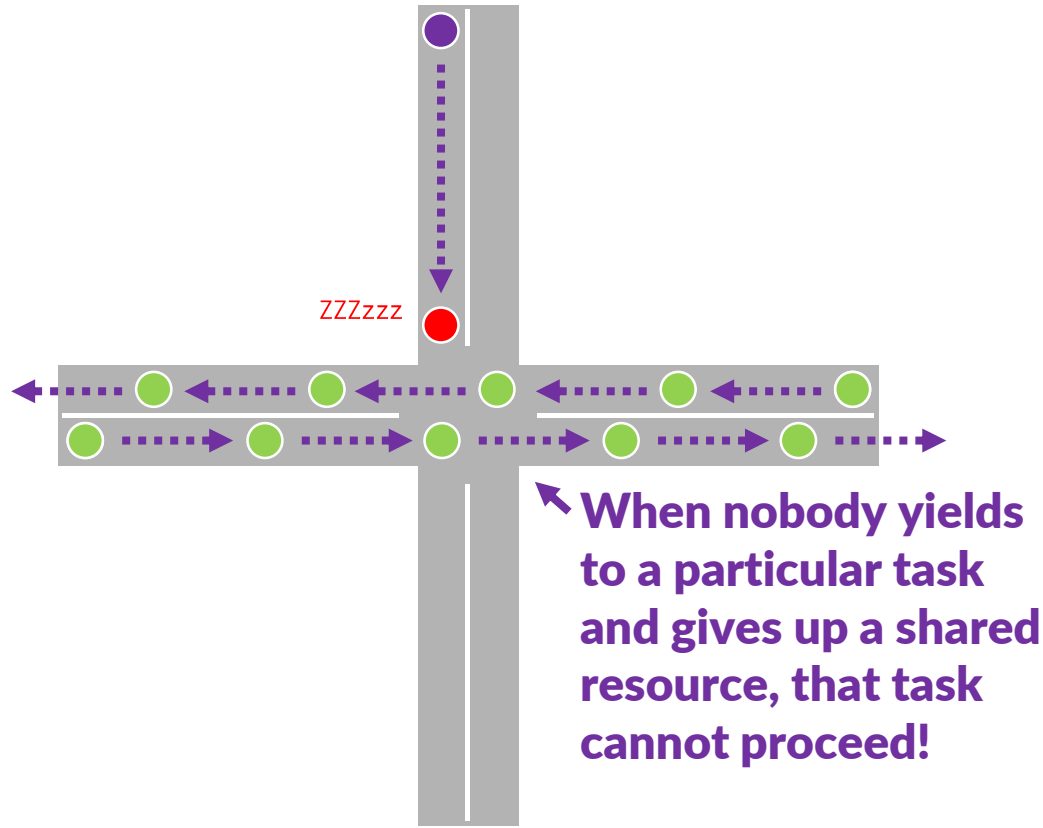
- However, in some circumstances, several cars may reach the intersection at the same time.
- If there is no previously defined way to handle this, they all wait for the others to get out of the way.
  - Forever.
- **Deadlock** occurs when multiple tasks are waiting for each other, making no progress.

# Synchronization solves deadlock



- Deadlock is a bug that needs extra consideration to avoid.
- In this case, you need some method of making only some of the cars (tasks) wait, while letting others go.
  - Traffic light, perhaps
- Beyond defining order, synchronization helps avoid these types of logical errors.

# Starved for attention...

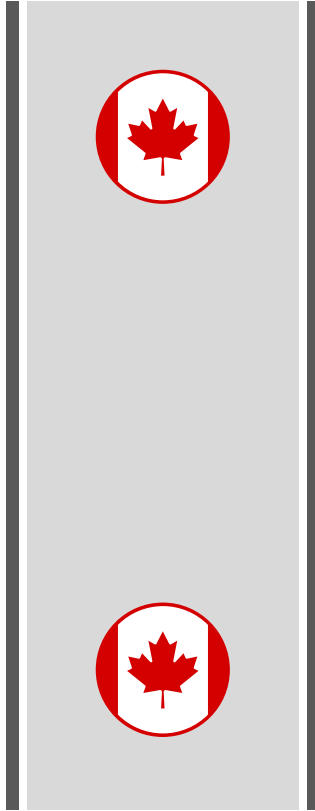


- Another issue, related to deadlock, is **starvation** where the system makes progress but one task is perpetually delayed.
- When some tasks have priority over resources, they may not give them up for other tasks.
  - Those tasks wait forever.
- Without a traffic light, you rely on people being nice. :(

# Starvation: a matter of fairness

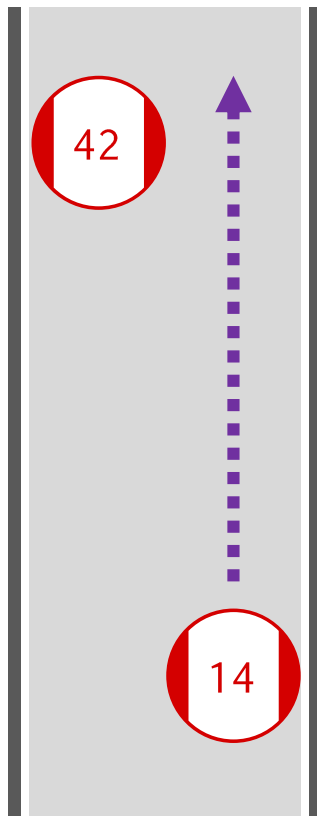
- This can happen in situations where “fairness” scheduling goes awry.
- If you have a webserver, the OS might schedule that process whenever there is some incoming requests.
  - What if you are getting a lot of traffic!
  - The OS might always schedule the webserver.
  - Important background tasks might not run!
- Preventing starvation might be keeping track of how much time a process has a resource and how long it has waited in line.
  - Low-priority tasks start at the back of the line and move up the queue the longer they wait... eventually cutting in front of high-priority tasks that start in the front.
- That's just *one* idea. Scheduling resources is a very difficult problem!

# Livelock: The hallway problem



- There is a narrow hallway.
- Let's say you have two very polite people.
- They walk toward each other... and try very hard to get out of each other's way.
  - They keep insisting the other go ahead of them.
- This is **livelock**, where two tasks are actively signaling the other to go and making no progress.

# Careful design works around livelock



- Livelock can be solved using a tie-breaking scheme.
  - Just find something comparable and unique among the tasks to create an arbitrary priority.
- All threads have IDs, so one easy strategy is to have the largest ID yield to the smaller.
  - This also helps starvation since livelock is starvation to the extreme: where everybody is starving.



# Deadlock vs. Livelock

- In deadlock, all tasks are waiting for a signal that will never happen.
  - They are *inactively* achieving nothing. (“ZZZZZzzzz”, “ZZZZZzzzz”)
- In contrast, livelock occurs when each task signals the other, and they respond by signaling back. (“No, you.” “No... you!”)
  - They are *actively* achieving nothing.
- Detecting that your program has a deadlock or livelock is tricky.
  - When it does, it may only happen a small percentage of the time.
  - In your OS course, you will learn more about deadlock detection and resolution.

# Solving things

- **Proper synchronization and planning can solve all these issues.**
  - Deadlock: Avoid patterns of critical sections that depend on each other.
  - Livelock: Establish a tie-breaking mechanism (thread with smallest ID goes first!)
  - Yet, it takes a good deal of programming experience to handle them.
- **The wide prevalence of multiprocessing and multithreading capable computers in the hands of average consumers is changing programming.**
  - New (and old) languages are being pushed for their better handling of concurrency issues.
  - Best-practices and frameworks continue to adapt to avoid many of the pitfalls we have discussed today.
  - Pay attention in your compilers and OS course to hone your own skill!

# pthread basic API summary

```
#include <pthread.h>
```

- Thread creation

- `int pthread_create(pthread_t*, pthread_attr_t*, void*(*)(void*), void*);`

- Join threads (wait until complete)

- `pthread_join(pthread_t, void**);` Waits for the given thread to end.

- Getting thread ID

- `pthread_t pthread_self();` Returns the thread ID of the current thread.

- Thread destruction (explicit)

- `pthread_cancel(pthread_t);` Attempts to preemptively exit the given thread.
  - `pthread_exit(void*);` Ends current thread and returns the provided value.

# pthread synchronization API summary

- Semaphores

- `#include <semaphore.h>`

- `int sem_init(sem_t*, 0, unsigned int initial_value);`

- Creates a semaphore with the given initial value. (The second argument means if the semaphore data is in shared memory. If non-zero, it can't be seen by other threads.)

- `int sem_wait(sem_t*);` Decrements counter unless it is 0 in which case it waits.

- `int sem_post(sem_t*);` Increments counter.

- Mutexes

- `#include <pthread.h>`

- `int pthread_mutex_init(pthread_mutex_t, NULL);` Creates a mutex (unlocked).

- `int pthread_mutex_lock(pthread_mutex_t*);` Waits until it can lock the mutex.

- `int pthread_mutex_unlock(pthread_mutex_t*);` Unlocks the mutex.

# Summary

- Threads are a different way to provide concurrency in a program.
  - Unlike process-level concurrency, threads share memory within the process.
- Synchronization primitives such as semaphores allow for creation of critical sections; necessary for correct concurrent code.
- Incorrect code may result in a new set of logical errors.
  - Race conditions – When execution order stochastically results in wrong behavior.
  - Deadlock – When resources are contended so much the program freezes.
  - Starvation – When a resource is greedily kept by a task, certain tasks freeze.
  - Livelock – Starvation happens at every task... they all actively yield to each other.