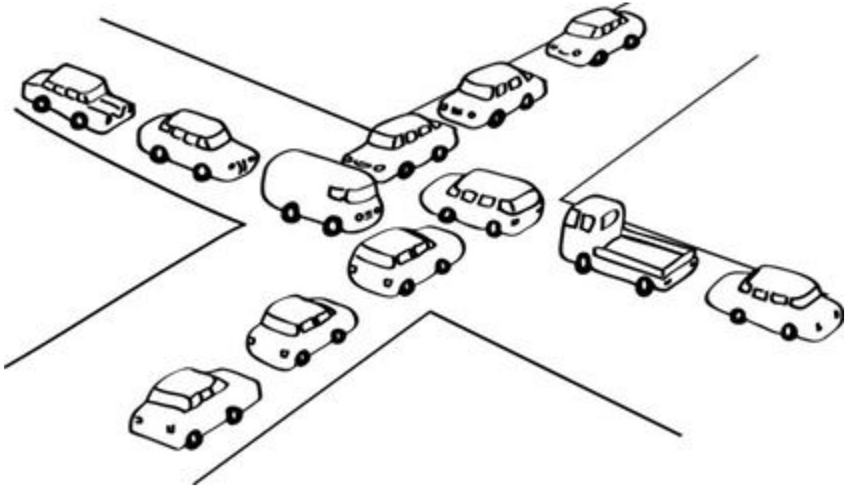
MONTANA
STATE UNIVERSITY

Concurrency In C

Doing Things… At The Same Time

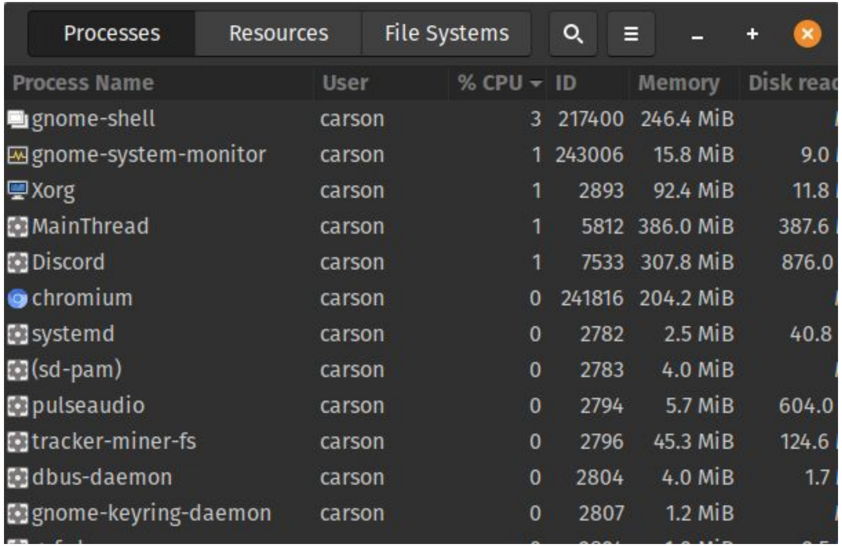
Concurrency

- Concurrency is the term we use for when logical operations overlap in time
- Concurrency is found at many layers in the computer



Concurrency

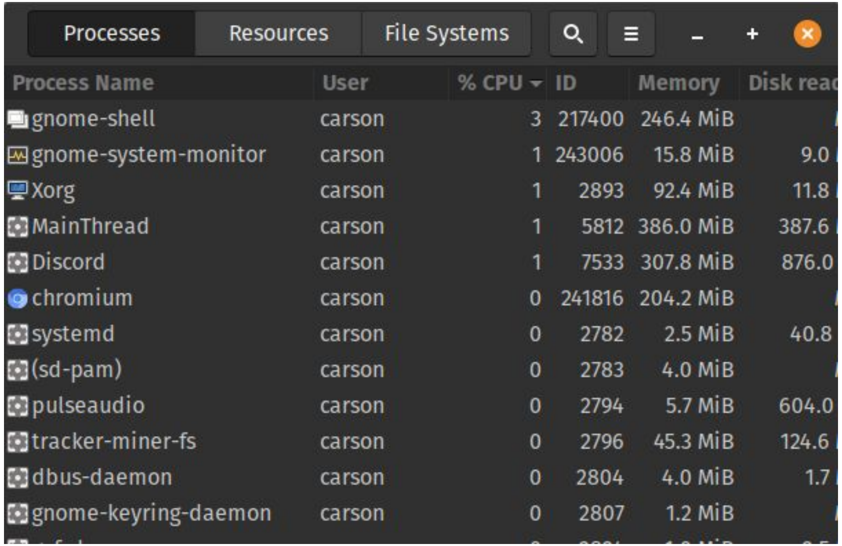
- The most obvious instance of concurrency on computers is *application concurrency*
- We all run multiple applications at the same time on our computers, phones, etc.



Process Name	User	% CPU	ID	Memory	Disk read
gnome-shell	carson	3	217400	246.4 MiB	
gnome-system-monitor	carson	1	243006	15.8 MiB	9.0
Xorg	carson	1	2893	92.4 MiB	11.8
MainThread	carson	1	5812	386.0 MiB	387.6
Discord	carson	1	7533	307.8 MiB	876.0
chromium	carson	0	241816	204.2 MiB	
systemd	carson	0	2782	2.5 MiB	40.8
(sd-pam)	carson	0	2783	4.0 MiB	
pulseaudio	carson	0	2794	5.7 MiB	604.0
tracker-miner-fs	carson	0	2796	45.3 MiB	124.6
dbus-daemon	carson	0	2804	4.0 MiB	1.7
gnome-keyring-daemon	carson	0	2807	1.2 MiB	

Concurrency

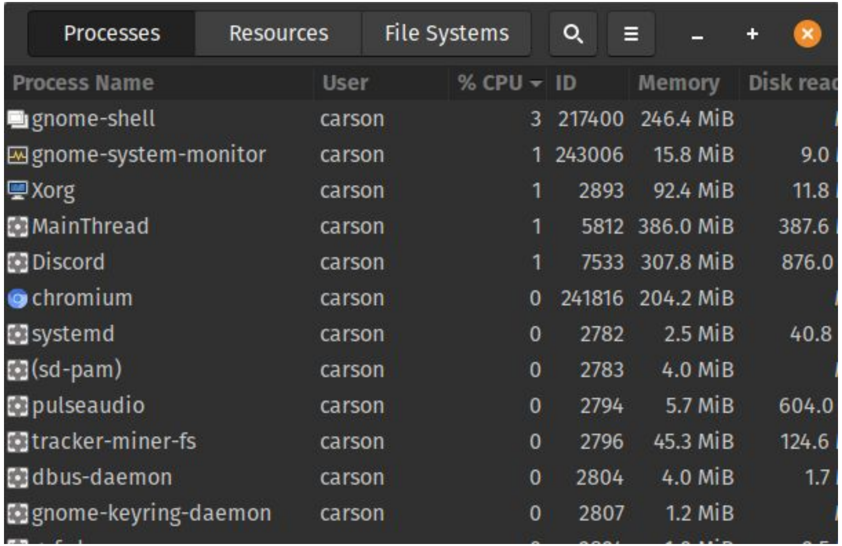
- Even on a single CPU, multiple programs can be run concurrently
- When one program *blocks* on an I/O operation, another can be run



Process Name	User	% CPU	ID	Memory	Disk read
gnome-shell	carson	3	217400	246.4 MiB	
gnome-system-monitor	carson	1	243006	15.8 MiB	9.0
Xorg	carson	1	2893	92.4 MiB	11.8
MainThread	carson	1	5812	386.0 MiB	387.6
Discord	carson	1	7533	307.8 MiB	876.0
chromium	carson	0	241816	204.2 MiB	
systemd	carson	0	2782	2.5 MiB	40.8
(sd-pam)	carson	0	2783	4.0 MiB	
pulseaudio	carson	0	2794	5.7 MiB	604.0
tracker-miner-fs	carson	0	2796	45.3 MiB	124.6
dbus-daemon	carson	0	2804	4.0 MiB	1.7
gnome-keyring-daemon	carson	0	2807	1.2 MiB	

Concurrency

- There are three main mechanism for building concurrent programs on modern CPUs:
  - Processes
  - Threads
  - I/O Multiplexing



Processes | Resources | File Systems

Process Name	User	% CPU	ID	Memory	Disk read
gnome-shell	carson	3	217400	246.4 MiB	
gnome-system-monitor	carson	1	243006	15.8 MiB	9.0
Xorg	carson	1	2893	92.4 MiB	11.8
MainThread	carson	1	5812	386.0 MiB	387.6
Discord	carson	1	7533	307.8 MiB	876.0
chromium	carson	0	241816	204.2 MiB	
systemd	carson	0	2782	2.5 MiB	40.8
(sd-pam)	carson	0	2783	4.0 MiB	
pulseaudio	carson	0	2794	5.7 MiB	604.0
tracker-miner-fs	carson	0	2796	45.3 MiB	124.6
dbus-daemon	carson	0	2804	4.0 MiB	1.7
gnome-keyring-daemon	carson	0	2807	1.2 MiB	

Processes

- The simplest way to create concurrent programs is via processes
- Use the venerable *fork()* function to create a new process
- *__pid_t* is an integer value

```
int main() {

    __pid_t pid = fork();
```

Processes

- __pid_t value
  - Negative Value: unable to create a child process (Out of Memory, etc.)
  - Zero: the newly created process
  - Positive: the original process gets this as the Process ID (PID) of the child process

```
int main() {

    __pid_t pid = fork();
```

Processes

- Both processes have a complete *copy of memory*
- This includes any files, sockets, etc. that are open
- The parent process must close any resources (e.g. open files) shared with the child or risk leaking them

```
int main() {

    __pid_t pid = fork();
```

Processes

- The memory spaces, however, are separate
- No shared *memory* between the processes, so no chance of corrupting one another's memory
- But also no way to communicate with other process

```
int main() {

    __pid_t pid = fork();
```

Processes

- Process Pros
  - Simple
  - Time Tested
- Process Cons
  - Heavyweight
  - Awkward to communicate between processes

```
int main() {

    __pid_t pid = fork();
```

Threads

- A more advanced technique is to use *threads*
- Threads are a lightweight alternative to processes
- Threads *share the same memory space*

```
pthread_t tid1, tid2;
pthread_create(&tid1, attr: NULL, run_thread, arg: 1);
```

Threads

- The standard threading library in unix is *pthreads*
- Provides 60+ functions and for working with threads
- The core function is *pthread_create()* which creates a new thread

```
pthread_t tid1, tid2;
pthread_create(&tid1, attr: NULL, run_thread, arg: 1);
```

Threads

- The arguments are
  - A pointer to a thread id (unsigned long)
  - An options argument
  - A *pointer* to the function to call as the root of the thread
  - An optional argument to pass to that function

```
pthread_t tid1, tid2;
pthread_create(&tid1, attr: NULL, run_thread, arg: 1);
```

Threads

- Thread termination
  - A thread can terminate implicitly by simply completing
  - It can terminate explicitly by calling the *pthread_exit()* function
  - It can be terminated by another thread with the *pthread_cancel()* function
  - Or the entire process can exit, killing all threads within it.

```
pthread_t tid1, tid2;
pthread_create(&tid1, attr: NULL, run_thread, arg: 1);
```

Threads

- Waiting on threads
- It is common to spawn threads and then wait for them to complete
- To do so, you can use the *pthread_join()* function
- A thread that joins another thread will wait until it completes before proceeding

```
pthread_create(&tid2, attr: NULL, run_thread, arg: 2);
pthread_join(tid2, thread_return: NULL);
```

Threads

- Threads Pros
  - Relatively fast and lightweight
  - Shared memory: easy thread communication!
- Threads Cons
  - More complex API
  - Shared memory: easy thread communication!

```
int main() {

    __pid_t pid = fork();
```

I/O Multiplexing

- The most complex way to achieve parallelism
- Uses the *select()* function
- Waits until one or more of a set of input sources is ready, or a timeout occurs
- Can be very efficient
- Very complex to deal with
    - We will not be using it in this class

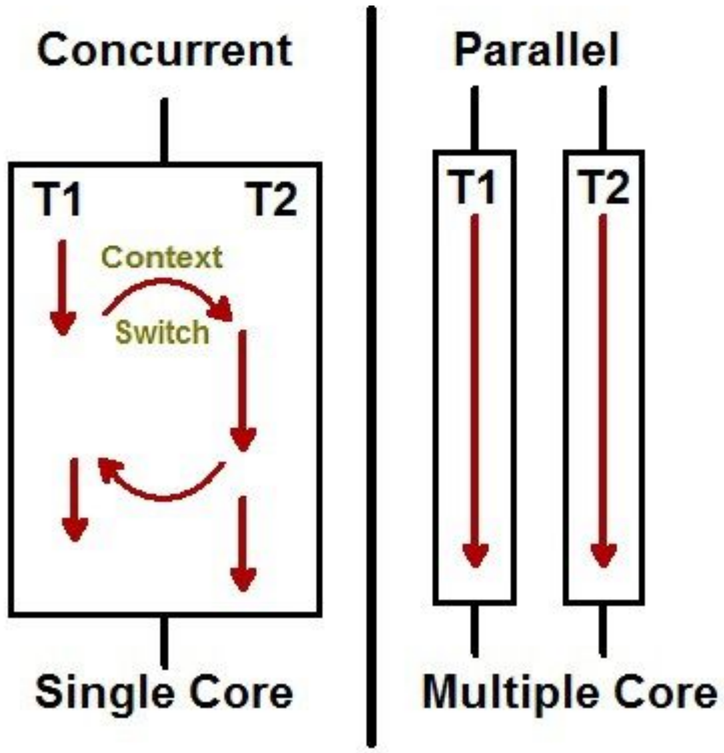
```
fd_set rfds;
struct timeval tv;
int retval;

/* Watch stdin (fd 0) to see when it has input. */
FD_ZERO( fdsetp: &rfds);
FD_SET( fd: 0,  fdsetp: &rfds);
/* Wait up to five seconds. */
tv.tv_sec = 5;
tv.tv_usec = 0;
retval = select( nfds: 1, &rfds,  writefds: NULL,  exceptfds: NULL, &tv);
/* Don't rely on the value of tv now! */

if (retval == -1)
    perror( s: "select()");
else if (retval)
    printf( format: "Data is available now.\n");
    /* FD_ISSET(0, &rfds) will be true. */
else
    printf( format: "No data within five seconds.\n");
```

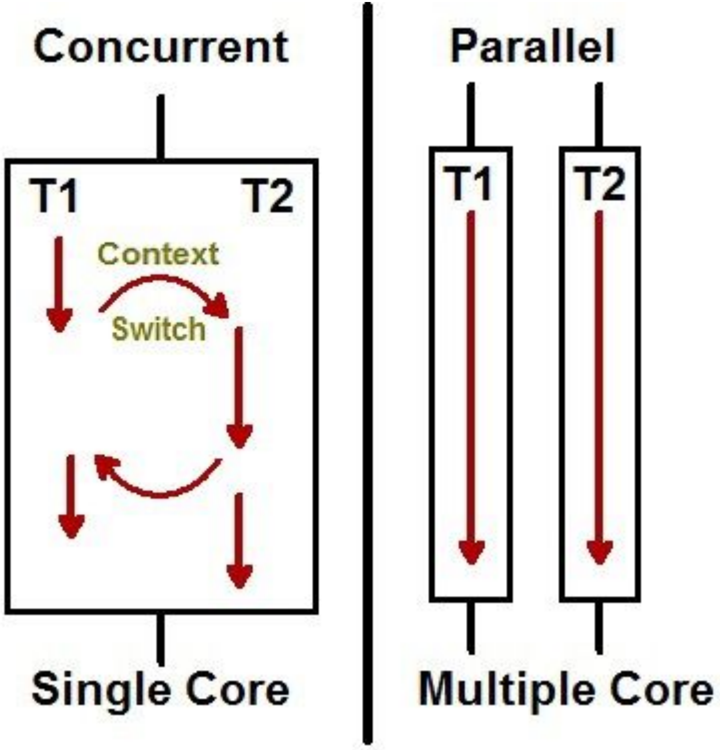
Concurrency vs Parallelism

- We have been talking about concurrency
- But you are probably *thinking* about parallelism
- They are *not* the same thing
- It is possible to be concurrent and not parallel
    - e.g. a single threaded core



Concurrency vs Parallelism

- On modern computer systems, concurrency almost always implies parallelism



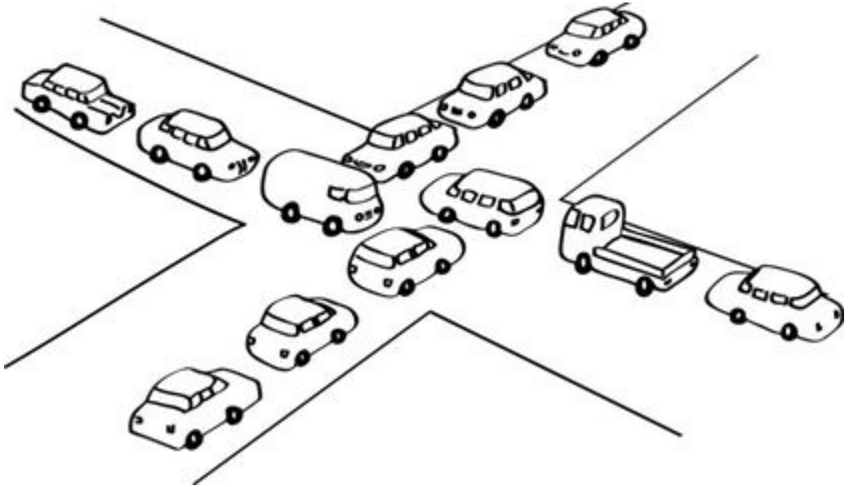
Synchronization

•••

Doing Things… At The Same Time. Safely.

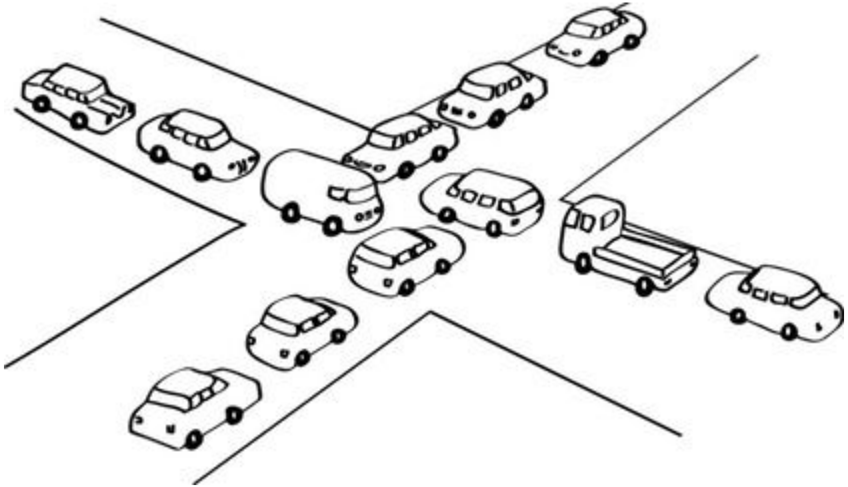
Concurrency

- If there is concurrent activity on a shared resource, you need to coordinate activities to avoid issues
- A traffic light coordinates access to a shared intersection



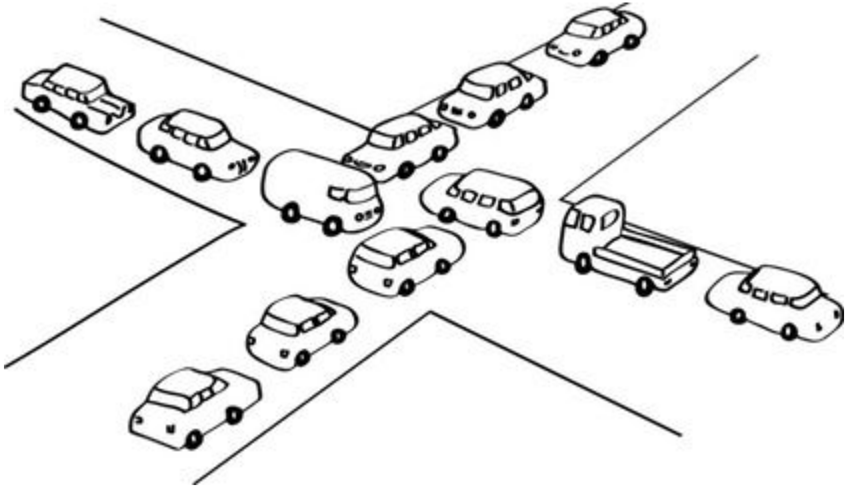
Concurrency

- The same logic applies to concurrency in computing systems
- Processes, since they do not share memory space, are less concerned with this issue
- Threads, however, since they share memory, must be very careful



Concurrency

- We will discuss two different tools for dealing with concurrency
- *Mutexes* - A lock to isolate critical parts of the code to a single thread
- *Semaphores* - A way to coordinate cooperation between threads



Mutexes

- A mutex is a simple lock with two primary operations
- *lock() (sometimes wait())*
  - Acquires the lock. If the lock is already held by someone else, the thread halts until it is released.
- *unlock() (sometimes signal())*
  - Releases the lock. If a thread is waiting on the lock, it is activated.

Mutexes

- In pthreads these functions are
    - *pthread_mutex_lock()*
    - *pthread_mutex_unlock()*
- The mutex is a struct which must be initialized and cleaned up
    - *pthread_mutex_init()*
    - *pthread_mutex_destory()*

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, mutexattr: NULL);
pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);
pthread_mutex_destroy(&lock);
```

Mutexes

- With a mutex, you want place the locking and unlocking around the *critical section* of your code
- The critical section is where *shared data* is being accessed or updated by multiple threads
- Demo...

```
pthread_mutex_lock(&lock);

/* Critical sectoion */
strcpy(buffer, src: "This buffer is accessed "
                    "by multiple threads");

pthread_mutex_unlock(&lock);
```

Mutexes

- So you can see how to use mutexes to ensure that only one thread at a time is updating shared data
- This will be required in your project as you take network input from two users concurrently

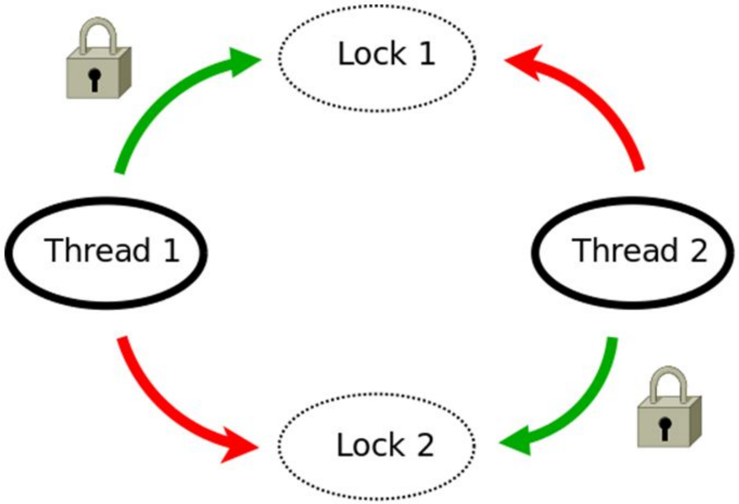
```
pthread_mutex_lock(&lock);

/* Critical sectoion */
strcpy(buffer, src: "This buffer is accessed "
                    "by multiple threads");

pthread_mutex_unlock(&lock);
```

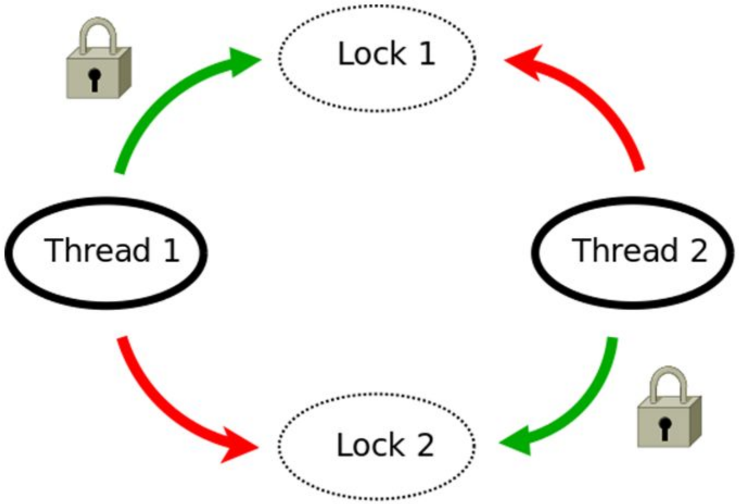
Mutexes - Deadlock

- What if
    - Thread 1 acquires lock 1
    - Thread 2 acquires lock 2
    - Thread 1 attempts to acquire lock 2
    - Thread 2 attempts to acquire lock 1
- Deadlock!



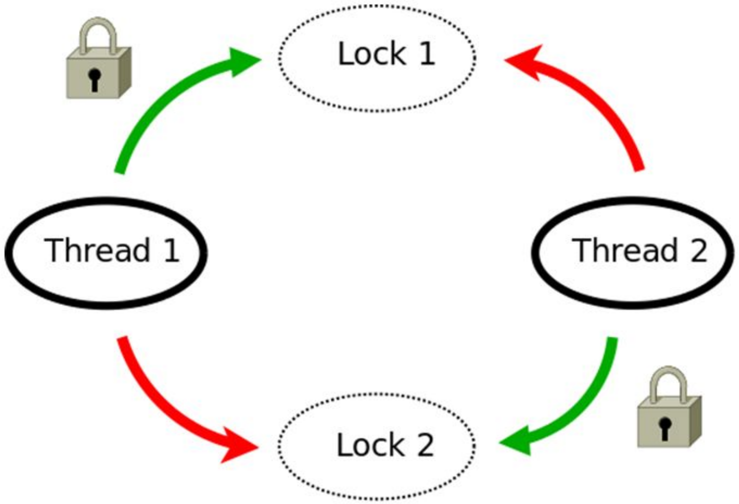
Mutexes - Deadlock

- To avoid deadlock you have a few options
  - Option 1: only one lock!
    - The Python GIL (Global Interpreter Lock)
    - Simple!
    - Destroys parallelism
  - Option 2: locks must be acquired in specific order
    - Better parallelism
    - Hard to enforce



Mutexes - Deadlock

- To avoid deadlock you have a few options
  - Option 3: All locks must be acquired up front, in a single atomic action
    - Better parallelism
    - Hard to know in advance what locks an operation might need

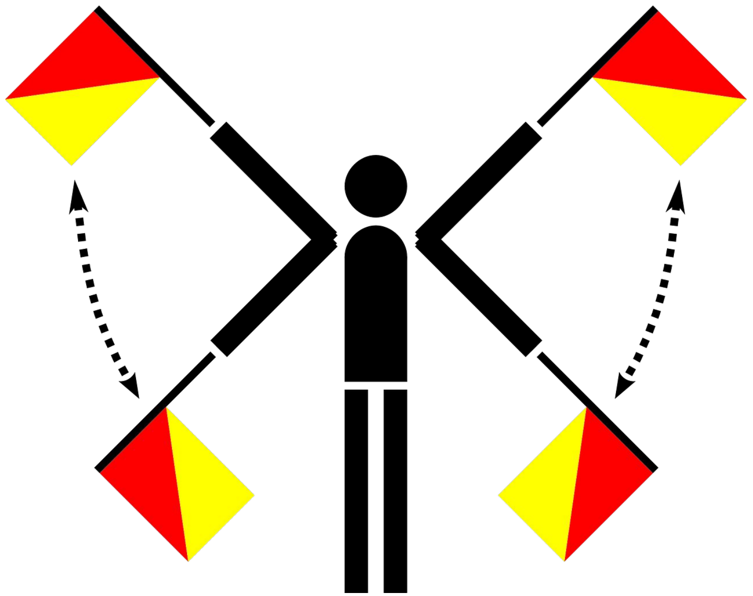


Mutexes - Deadlock

- For the project, I recommend option 1: a single lock
- If it’s good enough for python, it’s good enough for us

Semaphores

- Mutexes are about a single thread claiming and releasing sole ownership of a critical section
- Semaphores, on the other hand, are about multiple threads coordinating with one another



Semaphores

- Semaphores have a *counter* associated with them
- The core operations with a semaphore are
  - *wait()* - waits for the semaphore to be a non-0 value, then decrements it
  - *post()* - increments the semaphore counter

```
sem_t s;
sem_init(&s, pshared: 0, value: 1);
sem_wait(&s);
sem_post(&s);
sem_destroy(&s);
```

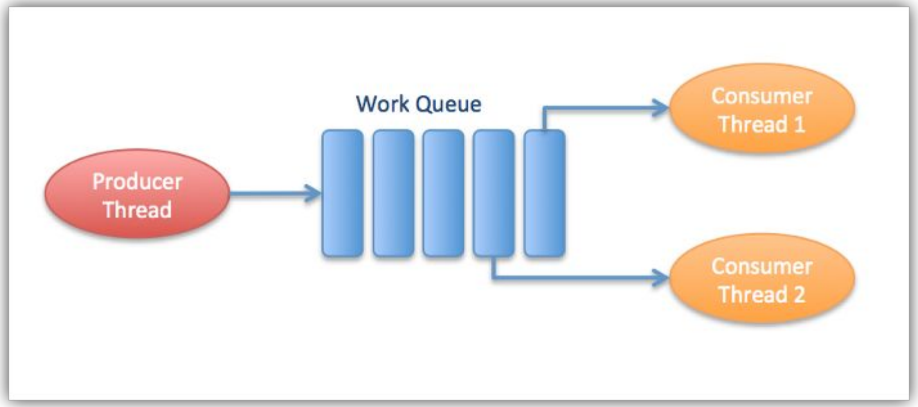
Semaphores

- A Semaphore with a limit of 1 is a *binary semaphore*
- Note that semaphores *do not* make any guarantees around shared data
- You may still need to do locking if there is shared data being access concurrently
- Demo...

```
sem_t s;
sem_init(&s, pshared: 0, value: 1);
sem_wait(&s);
sem_post(&s);
sem_destroy(&s);
```

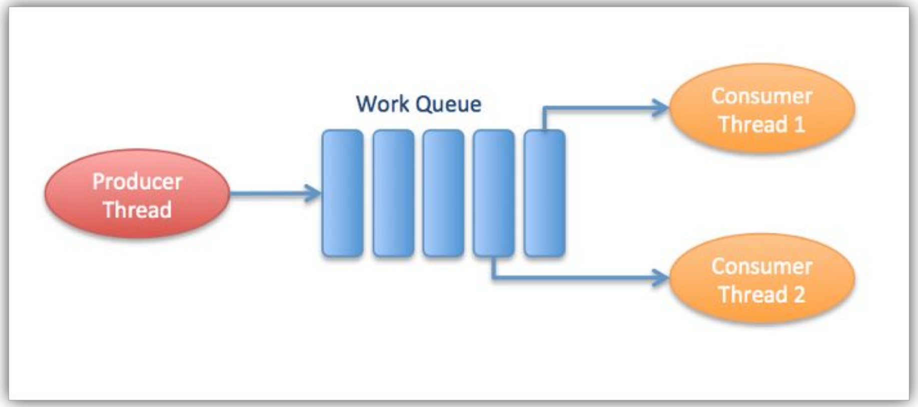
Semaphores

- Semaphores are useful for implementing things like *worker queues*
- Producer threads contribute work to a queue and signal
- Consumer threads wait for work to arrive
- Note that access to the shared Work Queue may still require locking!



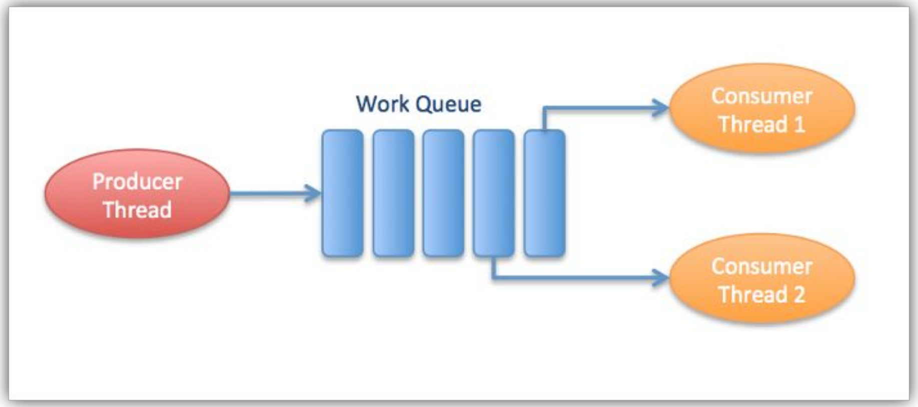
Semaphores

- Again, Semaphores are about thread coordination
- The thread signaling a semaphor is *not* a thread waiting on it
  - Compare w/ a mutex, where the thread unlocking the mutex better darn well be the thread that locked it!



Semaphores

- You can implement a lock with a binary semaphore by using conventions
  - Wait -> lock()
  - Signal -> unlock()
- But there is nothing enforcing the lock/unlock pairing as with a mutex



Review

- Concurrency is when two logical operations can overlap
- Concurrency and Parallelism are *not* the same thing
- The two major concurrency tools in C are
  - Processes - Heavy, Simple
  - Threads - Lightweight, Complex
- Concurrency with shared data requires coordination
- Mutexes (Locks) can be used to protect critical shared data
- Semaphores can be used to coordinate activities between threads but are not typically used for protecting critical shared data

MONTANA
STATE UNIVERSITY