

Introduction to Computer Organization

DIS 1A – Week 7

Slides modified from Eric Kim

Agenda

- Processes
- Threads
- Synchronization

Higher Order Parallelism

- With lower level parallelism (as in instruction level parallelism), we've reached something of a limit.
- That's not to say that you won't be able to squeeze more parallelism from the instructions.
- It's just that the work it takes to parallelize even further does not generally yield a proportional amount of performance benefit.

Task Level Parallelism

- Instead, we turn towards task level parallelism which isn't about finding parallelism in the instructions of a single flow of instruction executions, but rather multitasking.
- Multi-tasking means breaking down a program into separate tasks that can be executed concurrently, ideally on different processors (if relevant).

Processes

- The most difficult thing about processes: describing them in words.
- A “process” encompasses “the act of executing a flow of instructions”.
- So an executable program isn't really a process, but *executing* a program is.
- The OS maintains a list of these executions and these are what are known as “processes”

Processes

- Think of processes as having a similar level of independence as programs.
- When you open a terminal to run a program, you expect it to run in isolation compared to a program running on another terminal.
- This is the primary expectation placed on processes that are running in parallel.

Processes

- When it comes to processes (as with programs that are running concurrently), there is no guarantee as to the order in which the programs will execute.
- This leads to a complication in parallel programming which is synchronization.
- Specifically, you must make sure that your program works correctly, regardless of the order in which everything is executed.

Processes

- Each process thinks that it owns everything and that everything revolves around it.
 - The entire addressable memory space
 - [0x00000000000000, 0xFFFFFFFFFFFFFFFF]
 - All registers
 - The CPU
- In many ways, this means that using processes is simple and safe.
- If I'm running process A (say Chrome), I don't have to worry about it getting in the way of process B that is also running (say Netscape Navigator)

Processes

- Consider two programs A and B.
- At memory address 0x10, program A is storing a variable “int a”.
- If B accesses memory address 0x10, of course it's not going to find A's “int a”.
- Under this restriction, it's essentially impossible for a malicious process to tamper with the memory of another process.

Processes

- How to make a process:
 - `fork()`
- The `fork()` function will essentially clone the clone the existing program (memory and all) and after the point at which `fork` is called, two processes will be running.
- The original process that called `fork()` is considered the parent.
- The new process that was created is considered the child();

Processes

- After a process's creation the only difference between the parent and child processes is that the “child” will have 0 as the return value of `fork()`.
- The parent's return value for `fork` will be the pid of the child process
- Now previously, I said that a processes have the same level of autonomy as Chrome and Netscape Navigator.
- However, there is still a hierarchy to the processes that allow for some interaction.

Processes

- A process that creates another process (via fork) is considered the “parent”.
- In general, the parents have some level of control over the children processes.
- All processes maintain this hierarchy.
- A process can have an unbounded number of children but only one parent.

Processes

```
int main()
{
    int dummy = 0;
    if(fork() == 0)
    {
        dummy = 1;
    }
    else
    {
        dummy = 0;
    }
}
```

- When fork() is reached the child's value for fork() is 0
- Thus, the child will execute dummy = 1.
- Meanwhile, the parent's return value for fork is non-zero, therefore it will execute dummy = 0.

Processes

- How can they both have totally distinct address spaces?
- Think of `fork()` as having cloned the existing program and now you're running the original program and its clone. These two running processes are as distinct as Chrome and Netscape Navigator.
- So how is it that your computer can run multiple processes/programs simultaneously when each program hogs the entire CPU?

Processes

- Well, if you have multiple cores/CPU's, you can run different processes on each core.
- However, you're probably running more processes than you have CPU's on your computer.
- Essentially, what invariably has to happen is that the processes have to take turn sharing the CPU.

Processes

- This is done via:
 - Operating System Scheduling
 - Context Switches
- When it comes to Processes (and Threads), the OS is the king. It decides which process gets the processor and which one doesn't; it decides who lives and who dies.
- For instance, if Process A has been running for 10 ms. The CPU may be taken away from Process A and given it to Process B if it pleases the OS.

Processes

- This means that once multiple processes are launched, there is no guarantee as to the order in which the processes are executed.
- They are at the mercy of the almighty OS.
- But that's usually okay. You don't usually worry if Netscape Navigator is a few instructions ahead of Chrome.
- The OS will pick and choose whichever to execute.

Processes

- In order to do this, the OS has the responsibility of maintaining the state of each running process, also known as a “context”.
- The “context” of a process essentially consists of all of the stuff that the process believes it owns:
 - Values of registers
 - The stack
 - Page table
 - File table
 - Essentially, the process's identity.

Processes

- When the OS decides that Process A no longer amuses it and would like to perform a context switch to Process B it:
 - Saves Process A's current state/context (ie registers, stack, etc.)
 - Restores Process B's state.
- This idea of taking turns is known as “concurrency”. This is in contrast to parallelism, where things are actually performed simultaneously.

Processes

- Gee processes sure are swell.
- Because they're isolated and protected, they're simple to manage.
- You can run as many processes as you want and you don't have to worry about them messing each other up.
- However, are there situations in which processes are not ideal?

Processes

- The very thing that makes processes simple to use (isolation) is the thing that makes them inconvenient sometimes.
- What if you specifically want two processes to talk to each other or work together? You have to jump through hoops to get that to happen. For example, what if you just wanted two processes to share one measly int? Processes are completely blind to other process' memory spaces.
- The hoop: special OS managed inter-process communication (IPC)
- Additionally...

Processes

- Process Context Switching is an extremely expensive operation. Because Processes are so self contained, there is a lot that needs to be swapped over in a context switch.
- For these two reasons, if you want to do a computation where many executions are coordinating, processes may be too cumbersome and too slow.
- We need something lighter, faster, more dangerous...

Threads

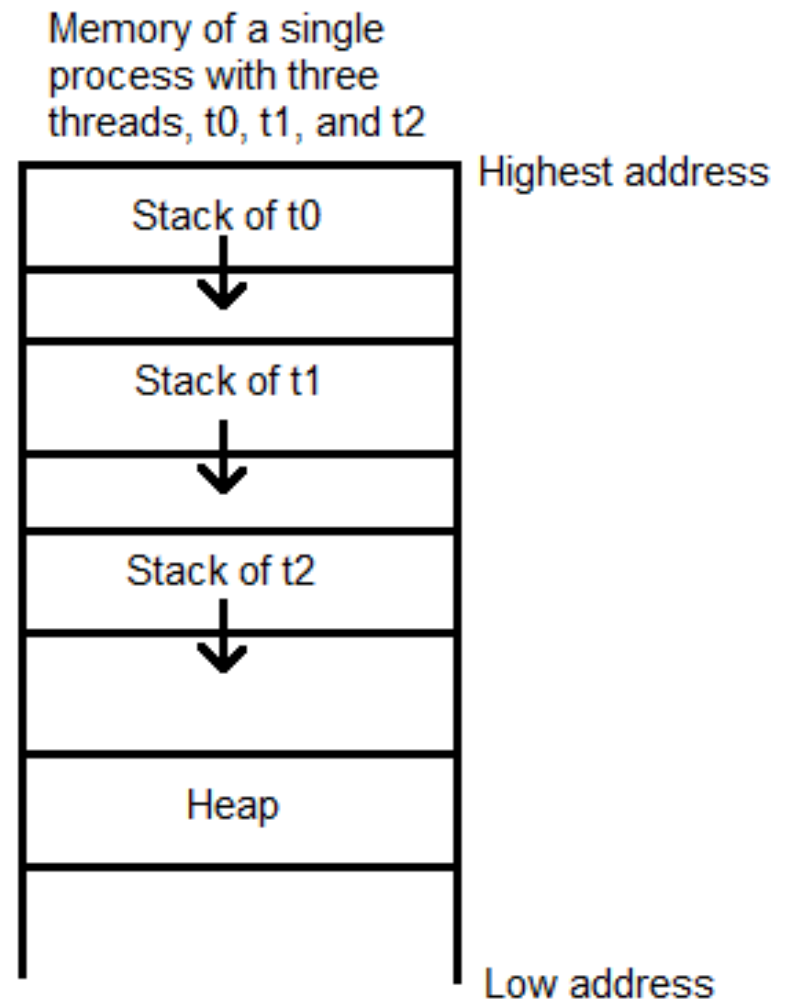
- Enter: Threads
- Threads are Process' leaner, lightweight siblings.
- Like processes, each thread runs its own distinct code in parallel.
- However, unlike processes, threads can easily acknowledge the existence of other threads.
- This is because threads share a single process and thus they share the resources of the process that they exist on.

Threads

- Each thread shares with all other threads on the process the entire addressable memory space
 - [0x000000000000, 0xFFFFFFFFFFFFFFF]
- Each thread has it's own space reserved in memory for its own stack.
- However, each thread also believes that it is the sole owner of:
 - CPU
 - All of the registers.

Threads

- Unlike with multiple processes, multiple threads on the same process share the same memory space. However, in an effort to allow for distinct execution, they each have space reserved in memory for their own stacks.



Threads

- Knowing this, what needs to be switched in a thread context switch?

Threads

- Knowing this, what needs to be switched in a thread context switch?
 - Pretty much just the registers.
 - The registers include %rip and %rsp, which means switching registers will account for the different instructions that threads will execute as well as the different stacks that each thread has.

Threads

- Threads have their own stack, but memory is shared. That means that among two threads running on the same process, the address 0x10 WILL point to the same thing.
- This is a major distinction from processes and can both be a major convenience or problem.
- A thread know where its own stack begins and ends, but there's nothing stopping it from accessing another thread's stack.

Threads

- As before, the thread's execution is at the mercy of the OS.
- However, this could present a problem.
- Generally when you use threads, you want them to work together in some way.
- You can't assume any particular order... unless you use special functions to manually maintain order.
- Another job for synchronization.

Threads

- How to create a thread:
- `int pthread_create(pthread_t *tid, pthread_attr_t const *attr, func *f, void *args)`
 - The argument `tid` is a thread id that is a pointer to a `pthread_t` passed in and assigned when the thread is created. A `pthread_t` is essentially a number
 - `attr` specifies options. By default, 0.
 - This creates a thread, assigns the thread id to `tid`. When the thread starts, it will call `f(args)`

Threads

- `pthread_t pthread_self(void);`

- Get own thread id.

`void pthread_exit(void *thread_return)`

- Thread kills itself with return value in `*thread_return`

Threads

- `int pthread_cancel(pthread_t tid)`
 - Thread murders another thread with id `tid`, in cold blood.
- `int pthread_join(pthread_t tid, void **thread_return)`
 - One thread waits for thread `tid` to complete before continuing.
 - Technically more reaping happens here.

Threads

- `int pthread_detach(pthread_t tid)`
 - Makes thread specified by `tid` autonomous. That is, the thread cannot be killed via `pthread_cancel` and it cannot be reaped/waited on by `pthread_join`.
 - The thread achieves invincibility.

Threads

Consider the following code:

```
int main()
{
    while(true)
    {
        pthread_t tid;
        int connfd = Accept(...);
        pthread_create(&tid, 0,
thread, &connfd);
    }
}
```

```
void *thread(void * args)
{
    pthread_detach(pthread_self());
    int * pconnfd = args;
    int connfd = *pconnfd;
    process(connfd);
    close(connfd);
    return 0;
}
```

- Does this work?

Threads

- connfd is a local variable of the main thread.
- Thus, connfd has an address that corresponds to some spot on the main.
- The value of connfd is assigned in the main thread by Accept.
- The address to this value of connfd is passed into the thread.

```
while(true)
{
    pthread_t tid;
    int connfd = Accept(...);
    pthread_create(&tid, 0,
thread, &connfd);
}
```

```
void *thread(void * args)
{
    ...
    int * pconnfd = args;
    int connfd = *pconnfd
    ...
}
```

Threads

- The thread then uses the address of the main thread's `connfd` for args.
- The thread will get the value and store it to a local variable.
- Meanwhile, the main thread re-assigns a value to `connfd`.
- The address that the old thread uses still corresponds to `connfd` in the main thread.

```
while(true)
{
    pthread_t tid;
    int connfd = Accept(...);
    pthread_create(&tid, 0,
thread, &connfd);
}
```

```
void *thread(void * args)
{
    ...
    int * pconnfd = args;
    int connfd = *pconnfd
    ...
}
```

Threads

- If that reassignment (`connfd = Accept(...)`) occurs before `connfd = *pconnfd`, then the old value will be wiped away.
- This is an example of a race condition (and a pretty bad one).
- Make sure this doesn't happen.

```
while(true)
{
    pthread_t tid;
    int connfd = Accept(...);
    pthread_create(&tid, 0,
        thread, &connfd);
}
```

```
void *thread(void * args)
{
    ...
    int * pconnfd = args;
    int connfd = *pconnfd
    ...
}
```

Threads

- Conclusion?
- Any time a thread gets a pointer that points to another thread's stack frame, be wary.
- We can fix this by making it so that the value passed into the thread is NOT a pointer pointing to the main thread's stack frame.

```
while(true)
{
    pthread_t tid;
    int connfd = Accept(...);
    pthread_create(&tid, 0,
thread, &connfd);
}
```

```
void *thread(void * args)
{
    ...
    int * pconnfd = args;
    int connfd = *pconnfd
    ...
}
```

Threads

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

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}

int main()
{
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, 0, thread_func, 0);
    pthread_create(&tid2, 0, thread_func, 0);
    pthread_join(tid1, 0);
    pthread_join(tid2, 0);
}
```

- In this example, the main thread will launch two threads with id's saved to tid1 and tid2. Each will run the function “thread_func()” and once they complete, each thread will terminate.
- Using pthread_join, the main thread will wait until tid1 and tid2 complete before finishing (more on that later).

Threads

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

```
int glob = 10;
```

```
void* proc_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

```
int main()
{
    fork();
    proc_func();
}
```

- This is a “Process” version of the same sort of code. This code will launch a process. Both the parent and child processes will call `proc_func()`.
- What will this print out?

Threads

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

```
int glob = 10;
```

```
void* proc_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

```
int main()
{
    fork();
    proc_func();
}
```

- Because these are processes, when you call fork, you clone the process and create a new process with a cloned memory space.
- Thus, they will both have “glob”, but they will be accessing different versions of glob.
- Output:
11
11

Threads

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

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

```
int main()
{
    pthread_t tid1;
    pthread_t tid2;
    pthread_create(&tid1, 0, thread_func, 0);
    pthread_create(&tid2, 0, thread_func, 0);
    pthread_join(tid1, 0);
    pthread_join(tid2, 0);
}
```

- What about the thread version?
- What will this print out?

Threads

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

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- There are two threads, t0 and t1 running this code.
- Since these are threads, they share the code and memory. Thus they both have a “glob” and this variable refers to the same variable.
- However, recall that there is no guarantee of the ordering

Threads

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

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- As a result, if the order of instructions was:
 1. t0: glob += 1
 2. t0: printf
 3. t1: glob += 1
 4. t1: printf
- The output would be:
11
12

Threads

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

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- However, if it was:
 1. t0: glob += 1
 2. t1: glob += 1
 3. t0: printf
 4. t1: printf
- The output would be:
12
12

Threads

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

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- Could it ever be:

12

11

or

11

11

?

Threads

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

int glob = 10;

void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- Surely not, right?
- We know from out-of-order processing that the order of the instructions is maintained even if execution doesn't strictly follow that ordering.
- Therefore, if a thread executes “printf”, it must have executed “glob += 1”. Thus:
11
11
- ...doesn't seem possible since when the second thread to reach printf calls printf, both threads must have called glob+=1 (if the first thread already reached printf)

Threads

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

```
int glob = 10;
```

```
void * thread_func()
{
    glob += 1;
    printf("%d\n", glob);
}
```

- Same with:

12

11

- If a thread prints out 12, that means both threads have already executed `glob += 1`
- Thus, if the first thread prints 12, then the next thread must print 12 since it's already executed `glob += 1`.
- Case closed. Right?

Threads

- For one thing, it's a fallacy to think in terms of the C functions.
- The ordering of instructions is a component of the assembly level.
- So what might the assembly code for `thread_func` look like?

Threads

- Assume a simplified version of printf that takes an int argument to print in %edi.

thread_func:

```
1. mov    glob(%rip),%eax
2. lea    0x1(%rax),%edi
3. mov    %edi,glob(%rip)
4. callq  printf
```

Threads

thread_func:

```
1. mov    glob(%rip),%eax
2. lea    0x1(%rax),%edi
3. mov    %edi,glob(%rip)
4. callq  printf
```

- If the order of execution is:
 1. t0: insn 1
 2. t1: insn 1
 3. t0: insn 2
 4. t1: insn 2
 5. t0: insn 3
 6. t1: insn 3
 7. t0: insn 4
 8. t1: insn 4
- Because both threads would load glob before either has committed a result to memory, the output would be:
11
11

Threads

thread_func:

```
1. mov    glob(%rip),%eax
2. lea    0x1(%rax),%edi
3. mov    %edi,glob(%rip)
4. callq  printf
```


- If the order of execution is:
 1. t0: insn 1
 2. t0: insn 2
 3. t0: insn 3
 4. t1: insn 1
 5. t1: insn 2
 6. t1: insn 3
 7. t1: insn 4
 8. t0: insn 4
- t0 increments glob and stores it so glob = 11. Then, t1 runs in its entirety, incrementing glob and printing out 12. Then, t0 prints the value of glob that it had stored in %edi, which was 11.
- Thus:

12
11

Note: gcc and threads

- To use pthreads in your C programs, add the "-lpthread" option to gcc command:

```
$ gcc -o mythread mythread.c -lpthread
```



Note: goes at end!

Example:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
void* threadjob(void *arg) {
    int* val = ((int*) arg);
    *val = 7;
    printf("  Thread finished.\n");
    return NULL;
}
int main() {
    pthread_t pth;
    int myval = 42;
    printf("  (1) myval is: %d\n", myval);
    pthread_create(&pth, NULL, threadjob,
    &myval);  pthread_join(pth, NULL);
    printf("  (2) myval is: %d\n", myval);
    return 0;
}
```

Question: What does program output?

Answer:

(1)myval is: 42
Thread finished.
(2) myval is: 7

Questions: Are there other possible outputs?

Answer:

No! pthread_join()
enforces a consistency.

Example:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
void* threadjob(void *arg) {
    int* val = ((int*) arg);
    *val = 7;
    printf("  Thread finished.\n");
    return NULL;
}
int main() {
    pthread_t pth;
    int myval = 42;
    printf("  (1) myval is: %d\n", myval);
    pthread_create(&pth, NULL, threadjob,
    printf("  (2) myval is: %d\n", myval);
    return 0;
}
```

Question: What are the possible outputs of the program?

Answer:

(1)myval is: 42	(1)myval is: 42
Thread finished.	(2)myval is: 42
(2) myval is: 7	Thread finished.

Suppose we removed that pthread_join() call...

Threads

- This was one of the simplest examples of shared memory, and its behavior could be completely wrong, simply because there's no guarantee of the order in which the assembly instructions would be executed.
- In order to properly use threads, we need synchronization.

Synchronization

- Threads allow a lightweight way to perform concurrency with shared variables
 - "With great power, comes great responsibility..."

Concurrency bugs: Program **sometimes** works, data is **sometimes** wrong, crashes **sometimes**...

Must carefully govern access to shared variables!



Synchronization + Race Conditions

- One of the major advantages of using threads over processes is the ease by which threads can share data.
- This of course, is marred by the potential that by using threads, you get the wrong result.
- That's bad.
- How do we ensure that behavior is correct, even when the execution order is not guaranteed?

Synchronization + Race Conditions

- We can enforce protection for shared variables or critical sections with semaphores, which are basically locks that tell us how many threads can enter a section of code.
- The semaphore is of type: `sem_t` and it is sort of a glorified counter or more conceptually, a door that allows/prevents entry into a section of code.

Synchronization + Race Conditions

- While the counter is non-zero, the door is open and thread can pass. When this happens, the counter decrements.
- When the counter is zero, the door is closed. The thread must wait for the door to be open again (counter becomes non-zero) before it can pass.

Synchronization + Race Conditions

- `sem_t sem;`
- `sem_wait(&sem)` – If `sem` is non-zero, return and decrement `sem` (the door is open, thread is allowed to pass). If `sem` is zero, wait until `sem` is non-zero, then decrement and return (the door is closed, wait for it to open).
- `sem_post(&sem)` – Increment `sem` by one. If `sem` was previously zero, the door was closed. Open the door and allow another thread in.

Example: Bounded Shared Buffer

- "Producer/Consumer" Scenario
- Application: Playing a video
 - Video decoder is constantly decoding frames and placing them in a buffer (ie each frame is an image)
 - Video player is constantly taking images from the buffer, and displaying them on the screen
- Guard access to buffer carefully

Synchronization + Race Conditions

- Say we have the following line of code:
 - `<line of code>`
- ...and we want to allow only two threads to be able to execute that line of code at any time.
- `sem_t sem;`
- `sem_init(&sem, 0, 2);` //0 is an options argument, 2 is the number of threads allowed
- `sem_wait(&sem);`
- `<line of code>;`
- `sem_post(&sem);`

Synchronization + Race Conditions

- Let's say we have some shared resource that you can read and write from. This can be done via two functions:

```
void read()  
{  
    <read shared resource>  
}
```

```
void write()  
{  
    <write shared resource>  
}
```


Synchronization + Race Conditions

- However, we want the following conditions:
 - There can be an unbounded number of concurrent readers.
 - There can be only 1 writer and if someone is writing, there can be no readers.
- How can we go about this?

Synchronization + Race Conditions

```
void read()  
{  
    <read>  
}
```

```
void write()  
{  
    <write>  
}
```

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
void read()
{
    sem_wait(&r);
    <read>
    sem_post(&r);
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- We'll definitely want some semaphores around the critical sections in read/write.
- Is this right?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
void read()
{
    sem_wait(&r);
    <read>
    sem_post(&r);
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- This prevents multiple writers, but it allows writing to happen at the same time as reading.
- Also, what do we initialize r to
We want unbounded readers.
- Let's try to solve the
“writing/reading at the same time” problem first

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
void read()
{
    sem_wait(&r)
    sem_wait(&w)
    <read>
    sem_wait(&w)
    sem_wait(&r)
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- Now, writing cannot happen at the same time as reading.
- What's the next step?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
void read()
{
    sem_wait(&r)
    sem_wait(&w)
    <read>
    sem_wait(&w)
    sem_wait(&r)
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- If we initialized r to, say 10, we could not have 10 readers reading at the same time because w is initialized to 1.
- We recognize that we only want to run `sem_wait(&w)` when there is at least one reader.

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
void read()
{
    sem_wait(&r)
    sem_wait(&w)
    <read>
    sem_wait(&w)
    sem_wait(&r)
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- If no readers, there might be a writer. Therefore, a reader must call `sem_post(&w)` to allow writer.
- If there are readers, then that means there cannot be a writer.
- Therefore, the reader needs to call `sem_wait(&w)`

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
void read()
{
    sem_wait(&r)
    if(there is one reader)
        sem_wait(&w)

    <read>

    if(this is the last reader)
        sem_post(&w)
        sem_post(&r)
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- Once a reader has finished reading, it must also check: if it is the last reader, then it should be able to allow a writer. Therefore it must release the lock by calling `sem_post(&w)`

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
int reader_count = 0;
void read()
{
    sem_wait(&r)
    reader_count++;
    if(reader_count == 1)
        sem_wait(&w)
```

<read>

```
    reader_count--;
    if(reader_count == 0)
        sem_post(&w)
    sem_post(&r)
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- Does this work?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
int reader_count = 0;
void read()
{
    sem_wait(&r)
    reader_count++;
    if(reader_count == 1)
        sem_wait(&w)

    <read>

    reader_count--;
    if(reader_count == 0)
        sem_post(&w)
        sem_post(&r)
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- Two problems, first of all, this only works if we can set r to infinity, which we can't.
- Second, there's still a possible race condition. What if two reader threads increment reader_count before either can get to reader_count == 1? The whole system messes up.

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w; // init to 1
int reader_count = 0;
void read()
{
    sem_wait(&r)
    reader_count++;
    if(reader_count == 1)
        sem_wait(&w)

    <read>

    reader_count--;
    if(reader_count == 0)
        sem_post(&w)
    sem_post(&r)
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- That this treats the <read> as the critical section, but that's not what we want to protect.
- We want to make sure that infinite readers can read, but only one reader can increment reader_count at a time.

Synchronization + Race Conditions

```
sem_t r; // init to 1
sem_t w; // init to 1
int reader_count = 0;
void read()
{
    sem_wait(&r);
    reader_count++;
    if(reader_count == 1)
        sem_wait(&w);
    sem_post(&r);
    <read>
    sem_wait(&r);
    reader_count--;
    if(reader_count == 0)
        sem_post(&w);
    sem_post(&r);
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- It satisfies the constraints (single writer, multiple reader).
- Is there maybe still a problem though?

Synchronization + Race Conditions

```
sem_t r; // init to 1
sem_t w; // init to 1
int reader_count = 0;
void read()
{
    sem_wait(&r);
    reader_count++;
    if(reader_count == 1)
        sem_wait(&w);
    sem_post(&r);
    <read>
    sem_wait(&r);
    reader_count--;
    if(reader_count == 0)
        sem_post(&w);
    sem_post(&r);
}
```

```
void write()
{
    sem_wait(&w);
    <write>
    sem_post(&w);
}
```

- This method is unfair to writers.
- That is to say, a potential writer could be starved since if readers keep coming in, the writer will never have a chance to write.