Concurrency, Races & Synchronization

CS 450: Operating Systems
Michael Lee <lee@iit.edu>
Agenda

- Concurrency: what, why, how
- Concurrency-related problems
- Locks & Locking strategies
- Concurrent programming with semaphores
§ Concurrency: what, why, how
concurrency = two or more overlapping execution contexts

execution context = a program and associated dynamic state
  (e.g., PC & stack)
parallelism, requiring multiple CPUs, is one way of realizing concurrency

i.e., computations run at the same time

concurrency can also be achieved with single CPU multiplexing

i.e., via context switches
parallelism

concurrency

context switch
Even on multi-CPU systems, CPU multiplexing is performed to achieve *higher levels of concurrency*
base unit of concurrency: *process*

- each execution context “owns” virtualized CPU, memory
- separate global address space
- *share-nothing* architecture
- context switches triggered by traps/interrupts
```c
int glob = 0;

int main() {
    pid_t pid;
    for (int i=0; i<5; i++) {
        if ((pid = fork()) == 0) {
            glob += 1;
            printf("Child %d glob = %d\n", i, glob);
            exit(0);
        } else {
            printf("Parent created child %d\n", pid);
        }
    }
    return 0;
}
```

Parent created child 97447
Parent created child 97448
Parent created child 97449
Child 1 glob = 1
Parent created child 97450
Child 2 glob = 1
Parent created child 97451
Child 4 glob = 1
Child 3 glob = 1
Child 0 glob = 1
Process model of concurrency provides system-level *sandboxing*

- separate processes cannot — by default — interfere with each other

- computations are performed entirely independently

- *interprocess communication* requires kernel APIs and data structures
within a process, default to a *single thread of execution*; i.e.,

- one path through program
- one stack
- blocking this thread (e.g., with I/O) blocks the entire process
but a single-threaded model is not always ideal or sufficient.
may desire *intra-process* concurrency!
why?

1. partition blocking activities
2. improve CPU utilization
3. performance gains from parallelization (most elusive!)
#1. Consider sequential operations that block on unrelated I/O resources

```c
read_from_disk1(buf1);  // block for input
read_from_disk2(buf2);  // block for input
read_from_network(buf3);  // block for input
process_input(buf1, buf2, buf3);
```

Would like to initiate input from separate blocking resources simultaneously
#2. consider interleaved but independent CPU & I/O operations

```c
while (1) {
    long_computation(); // CPU-intensive
    update_log_file(); // blocks on I/O
}
```

would like to start next computation while logging results from previous loop
#3. consider independent computations over large data set (software SIMD)

```c
int A[DIM][DIM], /* src matrix A */
    B[DIM][DIM], /* src matrix B */
    C[DIM][DIM]; /* dest matrix C */

/* C = A x B */
void matrix_mult () {
    int i, j, k;
    for (i=0; i<DIM; i++) {
        for (j=0; j<DIM; j++) {
            C[i][j] = 0;
            for (k=0; k<DIM; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

each cell in result is independent — need not serialize!
in each scenario, could make use of *multiple threads* within a *single process*

- permitted to independently block
- capable of running concurrently
- take advantage of global address space (i.e., easy sharing of data)
each thread needs to:

- share the global state (e.g., the code)
- track its own execution (e.g., on a stack)
- be given CPU time (i.e., be scheduled)
Global (shared)

Code

Data

Thread-local

Stack

Regs

context switch

$t_0$

$t_1$
but *who* (i.e., user or kernel) is responsible for tracking and scheduling threads?
option 1: *kernel (aka native)* threads

- kernel maintains metadata for 1 or more threads per process
- intra-process thread context switch is cheaper (why?) than process context switch, but still requires interrupt/trap
option 2: *user-space threads*

- kernel is only aware of “main” thread
- user code creates and tracks multiple thread states (e.g., stacks & register sets)
- context switches triggered by global timer or manually (cooperatively scheduled threads = “fibers”)

pros/cons?
kernel threads, pros:
  - thread parallelization is possible
  - process scheduler can be reused
  - no extra/duplicate work in user space

kernel threads, cons:
  - extra kernel metadata to manage
  - context switch requires trap/interrupt
user threads, pros:
- cheap to create and manage
- context switches are fast! (in user space)

user threads, cons:
- parallelization is not possible
- main thread blocks = all threads block
- replicating OS scheduler in user space
cooperatively-scheduled user threads, aka “fibers” can be even lighter weight!

- little to no scheduling overhead
- enables fine-grained, application-specific concurrency control
- may greatly reduce problems due to concurrency
option 3*: Hybrid threading

- M:N mapping of kernel to user threads
- User code responsible for scheduling tasks in system provided contexts
- Fast context switches + parallelizability, at cost of complexity (user & kernel)
Sample threading API: POSIX Threads — “pthreads”
/* thread creation */
int pthread_create (pthread_t *tid,
        const pthread_attr_t *attr,
        void *(*thread_fn)(void *),
        void *arg);

/* wait for termination; thread "reaping" */
int pthread_join (pthread_t tid,
        void **result_ptr);

/* terminates calling thread */
int pthread_exit (void *value_ptr);
int glob = 0;

void *inc_glob (void *num) {
    for (int i=0; i<10000; i++) {
        glob += 1;
    }
    printf("Thread %ld glob = %d\n", (int)num, glob);
    pthread_exit(NULL);
}

int main () {
    pthread_t tid;
    for (int i=0; i<5; i++){
        pthread_create(&tid, NULL, inc_glob, (void *)i);
        printf("Created thread %ld\n", (long)tid);
    }
    pthread_exit(NULL);
    return 0;
}
Note: pthreads API doesn’t specify whether implementation is kernel/user

- platform dependent

- most modern Unixes provide kernel-level threading support
Sample fiber library: libtask
swtch.com/libtask/
int glob = 0;

void inc_task (void *num) {
    for (int i=0; i<3; i++) {
        for (int j=0; j<10000; j++) {
            glob += 1;
        }
        printf("Task %d glob = %d\n", (int)num, glob);
        taskyield(); /* give up CPU */
    }
    taskexit(0);
}

/* note: libtask provides default main */
void taskmain(int argc, char **argv) {
    for (int i=0; i<5; i++) {
        taskcreate(inc_task,
               (void *)i,
               32768); /* stack size */
    }
}
```c
int
taskcreate(void (*fn)(void*), void *arg, uint stack)
{
    int id;
    Task *t;

    t = taskalloc(fn, arg, stack);
taskcount++;
id = t->id;
if(nalltask%64 == 0){
    alltask = realloc(alltask,
        (nalltask+64)*sizeof(alltask[0]));
if(alltask == nil){
    fprintf(stderr, "out of memory\n");
    abort();
}
}
t->alltaskslot = nalltask;
alltask[nalltask++] = t;
taskready(t);
return id;
}
```

```c
static Task*
taskalloc(void (*fn)(void*), void *arg, uint stack)
{
    Task *t;
sigset_t zero;
    uint x, y;
ulong z;

    /* allocate the task and stack together */
t = malloc(sizeof *t+stack);
if(t == nil){
    fprintf(stderr, "taskalloc malloc: %r\n");
    abort();
}
memset(t, 0,
    sizeof *t);
t->stk = (uchar*)(t+1);
t->stksize = stack;
t->id = ++taskidgen;
t->startfn = fn;
t->startarg = arg;

    /* do a reasonable initialization */
memset(&t->context.uc, 0,
    sizeof t->context.uc);
...
    /* must initialize with current context */
if(getcontext(&t->context.uc) < 0){
    fprintf(stderr, "getcontext: %r\n");
    abort();
}
...

    return t;
}
```
taskyield (and related) implementation is entirely in user-space (C & assembly)
- saves and restores task state (context) out of separately malloc’d stacks
- initiates coroutine jump (akin to setjmp/longjmp)
void contextswitch(Context *from, Context *to) {
    if(swapcontext(&from->uc, &to->uc) < 0){
        fprintf(2, "swapcontext failed: %r\n");
        assert(0);
    }
}

int swapcontext(ucontext_t *oucp, const ucontext_t *ucp) {
    if(getcontext(oucp) == 0)
        setcontext(ucp);
    return 0;
}

struct ucontext {
    sigset_t uc_sigmask;
    mcontext_t uc_mcontext;
    ...}

struct mcontext {
    ...
    int mc_ebp;
    ...
    int mc_ecx;
    int mc_eax;
    ...
    int mc_eip;
    int mc_cs;
    int mc_eflags;
    int mc_esp;
    ...}

#define setcontext(u) setmcontext(&(u)->uc_mcontext)
#define getcontext(u) getmcontext(&(u)->uc_mcontext)
#define SET setmcontext
#define GET getmcontext

SET:
    movl 4(%esp), %eax
    ...
    movl 28(%eax), %ebp
    ...
    movl 72(%eax), %esp
    pushl 60(%eax) /* new %eip */
    movl 48(%eax), %eax
    ret

GET:
    movl 4(%esp), %eax
    ...
    movl %ebp, 28(%eax)
    ...
    movl $1, 48(%eax) /* %eax */
    movl (%esp), %ecx /* %eip */
    movl %ecx, 60(%eax)
    leal 4(%esp), %ecx /* %esp */
    movl %ecx, 72(%eax)
    movl 44(%eax), %ecx /* restore %ecx */
    movl $0, %eax
    ret
Next: return to reason #3 for concurrency (performance)
int A[DIM][DIM], /* src matrix A */
B[DIM][DIM], /* src matrix B */
C[DIM][DIM]; /* dest matrix C */

/* C = A * B */
void matrix_mult () {
    int i, j, k;
    for (i=0; i<DIM; i++) {
        for (j=0; j<DIM; j++) {
            C[i][j] = 0;
            for (k=0; k<DIM; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}

Run time, with DIM=50,
500 iterations:

real 0m1.279s
user 0m1.260s
sys 0m0.012s
void run_with_thread_per_cell() {
    pthread_t ptd[DIM][DIM];
    int index[DIM][DIM][2];

    for(int i = 0; i < DIM; i++)
        for(int j = 0; j < DIM; j++) {
            index[i][j][0] = i;
            index[i][j][1] = j;
            pthread_create(&ptd[i][j], NULL,
                          row_dot_col,
                          index[i][j]);
        }

    for(i = 0; i < DIM; i++)
        for(j = 0; j < DIM; j++)
            pthread_join( ptd[i][j], NULL);
}

void row_dot_col(void *index) {
    int *pindex = (int *)index;
    int i = pindex[0];
    int j = pindex[1];

    C[i][j] = 0;
    for (int x=0; x<DIM; x++)
        C[i][j] += A[i][x]*B[x][j];
}

Run time, with DIM=50,
500 iterations:

real 4m18.013s
user 0m33.655s
sys 4m31.936s
void run_with_n_threads(int num_threads) {
    pthread_t tid[num_threads];
    int tdata[num_threads][2];
    int n_per_thread = DIM/num_threads;

    for (int i=0; i<num_threads; i++) {
        tdata[i][0] = i*n_per_thread;
        tdata[i][1] = (i < num_threads)? ((i+1)*n_per_thread)-1 : DIM;
        pthread_create(&tid[i], NULL, compute_rows, tdata[i]);
    }
    for (int i=0; i<num_threads; i++)
        pthread_join(tid[i], NULL);
}

void *compute_rows(void *arg) {
    int *bounds = (int *)arg;
    for (int i=bound[0]; i<=bound[1]; i++) {
        for (int j=0; j<DIM; j++) {
            C[i][j] = 0;
            for (int k=0; k<DIM; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
Dual processor system, kernel threading, DIM=50, 500 iterations
but matrix multiplication happens to be an *embarrassingly parallelizable* computation!

- not typical of concurrent tasks!
computations on shared data are typically *interdependent* (and this isn’t always obvious!)

— may impose a *cap* on parallelizability
Amdhal’s law predicts max speedup given two parameters:
- $P$: parallelizable fraction of program
- $N$: # of execution cores
max speedup $S = \frac{1}{\frac{P}{N} + (1 - P)}$

† $P \to 1; \ S \to N$

‡ $N \to \infty; \ S \to 1/(1 - P)$
Amdahl’s law is based on a **fixed problem size** with **fixed parallelizable fraction**

— but we can argue that as we have more computing power we simply tend to throw **larger / more granular problem sets** at it
e.g.,

- graphics processing: keep turning up resolution/detail
- weather modeling: increase model parameters/accuracy
- chess/weiqi AI: deeper search tree
Gustafson & Barsis posit that

- we tend to scale problem size to complete in the *same amount of time*, regardless of the number of cores
- parallelizable amount of work scales linearly with # of cores
Gustafson’s Law computes speedup based on:

- N cores

- non-parallelizable fraction, P
speedup \( S = N - P \cdot (N - 1) \)

\[ \uparrow P \rightarrow 1; \, S \rightarrow 1 \]
\[ \uparrow P \rightarrow 0; \, S \rightarrow N \]

- predicted speedup is \textit{linear} with respect to number of cores!
Amdahl’s vs. Gustafson’s:

- latter has rosier implications for big data / data science
  - but not all datasets naturally increase in resolution
- both stress the import of maximizing parallelization
some primary challenges of concurrent programming are to:

1. identify thread interdependencies
2. identify (1)’s potential ramifications
3. ensure correctness
e.g., final change in count? (expected = 2)

Thread A

```
  al  count = count + 1
```

Thread B

```
  bl  count = count + 1
```

interdependency: shared var \texttt{count}
factoring in machine-level granularity:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a1</strong> lw (count), %r0</td>
<td><strong>b1</strong> lw (count), %r0</td>
</tr>
<tr>
<td><strong>a2</strong> add $1, %r0</td>
<td><strong>b2</strong> add $1, %r0</td>
</tr>
<tr>
<td><strong>a3</strong> sw %r0, (count)</td>
<td><strong>b3</strong> sw %r0, (count)</td>
</tr>
</tbody>
</table>

answer: either +1 or +2!
race condition(s) exists when results are dependent on the order of execution of concurrent tasks
shared resource(s) are the problem
or, more specifically, concurrent mutability of shared resources
code that accesses shared resource(s) = critical section
synchronization:

time-sensitive coordination of critical sections so as to avoid race conditions
e.g., specific ordering of different threads, or mutually exclusive access to variables
important: try to separate and decouple application logic from synchronization details

- not doing this well adds unnecessary complexity to high-level code, and makes it much harder to test and maintain!
most common technique for implementing synchronization is via software “locks”

- explicitly required & released by consumers of shared resources
§ Locks & Locking Strategies
basic idea:

- create a shared software construct that has well defined concurrency semantics
  - aka. a “thread-safe” object
- Use this object as a guard for another, un-thread-safe shared resource
Thread A

```
   al  count = count + 1
```

Thread B

```
   bl  count = count + 1
```

**Diagram:**

- Thread A
- Thread B
- Count
- Locked resource

Thread A and Thread B both acquire the locked resource before incrementing the count.
Thread A

\[ a1 \text{ count} = \text{count} + 1 \]

Thread B

\[ b1 \text{ count} = \text{count} + 1 \]
Thread A

```plaintext
a1 count = count + 1
```

Thread B

```plaintext
b1 count = count + 1
```

Diagram:
- Thread A (`T_A`)
- Thread B (`T_B`)
- Mutual exclusion lock
- Count variable `count`
Thread A

\[ a_1 \text{ count} = \text{count} + 1 \]

Thread B

\[ b_1 \text{ count} = \text{count} + 1 \]
Thread A

```c
a1  count = count + 1
```

Thread B

```c
b1  count = count + 1
```

Count

allocated

use
locking can be:

- **global** *(coarse-grained)*

- **per-resource** *(fine-grained)*
coarse-grained locking policy
coarse-grained locking policy
coarse-grained locking policy
coarse-grained locking:
- is (typically) easier to reason about
- results in a lot of *lock contention*
- could result in *poor resource utilization* — may be impractical for this reason
fine-grained locking policy
fine-grained locking:
- may reduce (individual) lock contention
- may improve resource utilization
- can result in a lot of locking overhead
- can be much harder to verify correctness!
  - e.g., due to problems such as *deadlock*
deadlock with fine-grained locking policy
so far, have only considered *mutual exclusion*

what about instances where we require a *specific order* of execution?

- often very difficult to achieve with simple-minded locks
§ Abstraction: Semaphore
The Little Book of Semaphores
Second Edition
Allen B. Downey

Download the book in PDF now!

The Little Book of Semaphores is a free (in both senses of the word) textbook that introduces the principles of synchronization for concurrent programming.

In most computer science curricula, synchronization is a module in an Operating Systems class. OS textbooks present a standard set of problems with a standard set of solutions, but most students don't get a good understanding of the material or the ability to solve similar problems.
Chapter 2

Semaphores

In real life a semaphore is a system of signals used to communicate visually, usually with flags, lights, or some other mechanism. In software, a semaphore is a data structure that is useful for solving a variety of synchronization problems.

Semaphores were invented by Edsger Dijkstra, a famously eccentric computer scientist. Some of the details have changed since the original design, but the basic idea is the same.

2.1 Definition

A semaphore is like an integer, with three differences:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

Semaphore rules:

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.
Semaphores

• In general, there is no way to know before a thread decrements a semaphore whether it will block or not (in specific cases you might be able to prove that it will or will not).

• After a thread increments a semaphore and another thread gets woken up, both threads continue running concurrently. There is no way to know which thread, if either, will continue immediately.

• When you signal a semaphore, you don't necessarily know whether another thread is waiting, so the number of unblocked threads may be zero or one.

Finally, you might want to think about what the value of the semaphore means. If the value is positive, then it represents the number of threads that can decrement without blocking. If it is negative, then it represents the number of threads that have blocked and are waiting. If the value is zero, it means there are no threads waiting, but if a thread tries to decrement, it will block.

2.2 Syntax

In most programming environments, an implementation of semaphores is available as part of the programming language or the operating system. Different implementations sometimes offer slightly different capabilities, and usually require different syntax.

In this book I will use a simple pseudo-language to demonstrate how semaphores work. The syntax for creating a new semaphore and initializing it is

Listing 2.1: Semaphore initialization syntax

1 fred = Semaphore(1)

The function Semaphore is a constructor; it creates and returns a new Semaphore. The initial value of the semaphore is passed as a parameter to the constructor.

The semaphore operations go by different names in different environments. The most common alternatives are

Listing 2.2: Semaphore operations

1 fred.increment()
2 fred.decrement()

and

Listing 2.3: Semaphore operations

1 fred.signal()
2 fred.wait()
## Operation names?

<table>
<thead>
<tr>
<th></th>
<th>fred.increment_and_wake_a_waiting_process_if_any()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.decrement_and_block_if_the_result_is_negative()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>fred.increment()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.decrement()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>fred.signal()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.wait()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>fred.V()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fred.P()</td>
</tr>
</tbody>
</table>
How to use semaphores for synchronization?

1. Identify essential usage “patterns”
2. Solve “classic” synchronization problems
Essential synchronization criteria:

1. avoid *starvation*
2. guarantee *bounded waiting*
3. no assumptions on *relative speed* (of threads)
4. allow for *maximum concurrency*
§ Using Semaphores for Synchronization
Basic patterns:

I. Rendezvous
II. Mutual exclusion (Mutex)
III. Multiplex
IV. Generalized rendezvous / Barrier & Turnstile
I. Rendezvous

Ensure that $a_1 < b_2$, $b_1 < a_2$
3.2 Rendezvous

3.2.2 Rendezvous solution
Here is my solution, based on the previous hint:

Thread A

1. statement a1
2. aArrived.signal()
3. bArrived.wait()
4. statement a2

Thread B

1. statement b1
2. bArrived.signal()
3. aArrived.wait()
4. statement b2

While working on the previous problem, you might have tried something like this:

Thread A

1. statement a1
2. bArrived.wait()
3. aArrived.signal()
4. statement a2

Thread B

1. statement b1
2. aArrived.wait()
3. bArrived.signal()
4. statement b2

This solution also works, although it is probably less efficient, since it might have to switch between A and B one time more than necessary.

If A arrives first, it waits for B. When B arrives, it wakes A and might proceed immediately to its wait in which case it blocks, allowing A to reach its signal, after which both threads can proceed.

Think about the other possible paths through this code and convince yourself that in all cases neither thread can proceed until both have arrived.

3.2.3 Deadlock #1
Again, while working on the previous problem, you might have tried something like this:

Thread A

1. statement a1
2. bArrived.wait()
3. aArrived.signal()
4. statement a2

Thread B

1. statement b1
2. aArrived.wait()
3. bArrived.signal()
4. statement b2

If so, I hope you rejected it quickly, because it has a serious problem. Assuming that A arrives first, it will block at its wait. When B arrives, it will also block, since A wasn’t able to signal aArrived. At this point, neither thread can proceed, and never will.

This situation is called a deadlock and, obviously, it is not a successful solution of the synchronization problem. In this case, the error is obvious, but often the possibility of deadlock is more subtle. We will see more examples later.

```
aArrived = Semaphore(0)
bArrived = Semaphore(0)
```
Note: Swapping 2 & 3 → Deadlock!

Each thread is waiting for a signal that will never arrive
Basic synchronization patterns

3.3 Mutex

A second common use for semaphores is to enforce mutual exclusion. We have already seen one use for mutual exclusion, controlling concurrent access to shared variables. The mutex guarantees that only one thread accesses the shared variable at a time.

A mutex is like a token that passes from one thread to another, allowing one thread at a time to proceed. For example, in *The Lord of the Flies* a group of children use a conch as a mutex. In order to speak, you have to hold the conch. As long as only one child holds the conch, only one can speak.

Similarly, in order for a thread to access a shared variable, it has to “get” the mutex; when it is done, it “releases” the mutex. Only one thread can hold the mutex at a time.

Puzzle: Add semaphores to the following example to enforce mutual exclusion to the shared variable *count*.

Thread A

\[
\text{count} = \text{count} + 1
\]

Thread B

\[
\text{count} = \text{count} + 1
\]

Ensure that critical sections do not overlap
3.3.2 Mutual exclusion solution

Here is a solution:

Thread A

mutex.wait()
# critical section
count = count + 1
mutex.signal()

Thread B

mutex.wait()
# critical section
count = count + 1
mutex.signal()

Since mutex is initially 1, whichever thread gets to the wait first will be able to proceed immediately. Of course, the act of waiting on the semaphore has the effect of decrementing it, so the second thread to arrive will have to wait until the first signals.

I have indented the update operation to show that it is contained within the mutex. In this example, both threads are running the same code. This is sometimes called a symmetric solution. If the threads have to run different code, the solution is asymmetric. Symmetric solutions are often easier to generalize. In this case, the mutex solution can handle any number of concurrent threads without modification. As long as every thread waits before performing an update and signals after, then no two threads will access count concurrently.

Often the code that needs to be protected is called the critical section, I suppose because it is critically important to prevent concurrent access. In the tradition of computer science and mixed metaphors, there are several other ways people sometimes talk about mutexes. In the metaphor we have been using so far, the mutex is a token that is passed from one thread to another. In an alternative metaphor, we think of the critical section as a room, and only one thread is allowed to be in the room at a time. In this metaphor, mutexes are called locks, and a thread is said to lock the mutex before entering and unlock it while exiting. Occasionally, though, people mix the metaphors and talk about “getting” or “releasing” a lock, which doesn’t make much sense. Both metaphors are potentially useful and potentially misleading. As you work on the next problem, try out both ways of thinking and see which one leads you to a solution.

3.4 Multiplex

Puzzle: Generalize the previous solution so that it allows multiple threads to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent threads. In other words, no more than \( n \) threads can run in the critical section at the same time.

This pattern is called a multiplex. In real life, the multiplex problem occurs at busy nightclubs where there is a maximum number of people allowed in the building at a time, either to maintain fire safety or to create the illusion of exclusivity.

```python
mutex = Semaphore(1)

Thread A
mutex.wait()
# critical section
count = count + 1
mutex.signal()

Thread B
mutex.wait()
# critical section
count = count + 1
mutex.signal()
```

_Danger:_ if a thread blocks while “holding” the mutex semaphore, it will also block all other mutex-ed threads!
To allow multiple threads to run in the critical section, just initialize the semaphore to \( n \), which is the maximum number of threads that should be allowed.

At any time, the value of the semaphore represents the number of additional threads that may enter. If the value is zero, then the next thread will block until one of the threads inside exits and signals. When all threads have exited the value of the semaphore is restored to \( n \).

Since the solution is symmetric, it's conventional to show only one copy of the code, but you should imagine multiple copies of the code running concurrently in multiple threads.

Listing 3.1: Multiplex solution

```
1 multiplex.wait()
2 critical section
3 multiplex.signal()
```

What happens if the critical section is occupied and more than one thread arrives? Of course, what we want is for all the arrivals to wait. This solution does exactly that. Each time an arrival joins the queue, the semaphore is decremented, so that the value of the semaphore (negated) represents the number of threads in queue.

When a thread leaves, it signals the semaphore, incrementing its value and allowing one of the waiting threads to proceed.

Thinking again of metaphors, in this case I find it useful to think of the semaphore as a set of tokens (rather than a lock). As each thread invokes \texttt{wait}, it picks up one of the tokens; when it invokes \texttt{signal} it releases one. Only a thread that holds a token can enter the room. If no tokens are available when a thread arrives, it waits until another thread releases one.

In real life, ticket windows sometimes use a system like this. They hand out tokens (sometimes poker chips) to customers in line. Each token allows the holder to buy a ticket.

III. multiplex = Semaphore(N)

Permits \( N \) threads through into their critical sections
To allow multiple threads to run in the critical section, just initialize the semaphore to $n$, which is the maximum number of threads that should be allowed. At any time, the value of the semaphore represents the number of additional threads that may enter. If the value is zero, then the next thread will block until one of the threads inside exits and signals. When all threads have exited the value of the semaphore is restored to $n$.

Since the solution is symmetric, it’s conventional to show only one copy of the code, but you should imagine multiple copies of the code running concurrently in multiple threads.

Listing 3.1: Multiplex solution

```
1 multiplex.wait()
2 critical section
3 multiplex.signal()
```

What happens if the critical section is occupied and more than one thread arrives? Of course, what we want is for all the arrivals to wait. This solution does exactly that. Each time an arrival joins the queue, the semaphore is decremented, so that the value of the semaphore (negated) represents the number of threads in queue. When a thread leaves, it signals the semaphore, incrementing its value and allowing one of the waiting threads to proceed.

Thinking again of metaphors, in this case I find it useful to think of the semaphore as a set of tokens (rather than a lock). As each thread invokes `wait`, it picks up one of the tokens; when it invokes `signal` it releases one. Only a thread that holds a token can enter the room. If no tokens are available when a thread arrives, it waits until another thread releases one.

In real life, ticket windows sometimes use a system like this. They hand out tokens (sometimes poker chips) to customers in line. Each token allows the holder to buy a ticket.

### IV. Generalized Rendezvous / Barrier

Puzzle: Generalize the rendezvous solution. Every thread should run the following code:

Listing 3.2: Barrier code

```
1 rendezvous
2 critical point
```
3.5 Barrier

3.5.1 Barrier hint

For many of the problems in this book I will provide hints by presenting the variables I used in my solution and explaining their roles.

Listing 3.3: Barrier hint

```java
1 n = the number of threads
2 count = 0
3 mutex = Semaphore(1)
4 barrier = Semaphore(0)
```

count keeps track of how many threads have arrived.

mutex provides exclusive access to count so that threads can increment it safely.

barrier is locked (zero or negative) until all threads arrive; then it should be unlocked (1 or more).

Hint:
1 rendezvous
2
3 mutex.wait()
4    count = count + 1
5 mutex.signal()
6
7 if count == n: barrier.signal()
8
9 barrier.wait()
10 barrier.signal()
11
12 critical point
1 rendezvous
2
3 mutex.wait()
4     count = count + 1
5 mutex.signal()
6
7 if count == n: turnstile.signal()
8
9 turnstile.wait()
10 turnstile.signal()
11
12 critical point

state of turnstile after all threads make it to 12?
1 rendezvous
2
3 mutex.wait()
4    count = count + 1
5    if count == n: turnstile.signal()
6 mutex.signal()
7
8 turnstile.wait()
9 turnstile.signal()
10
11 critical point

fix for non-determinism (but still off by one)
next: would like a **reusable** barrier
need to **re-lock** turnstile
This attempt fixes the previous error, but a subtle problem remains.

Listing 3.8: Reusable barrier non-solution

```c
1 rendezvous
2
3 mutex.wait()
4     count += 1
5     if count == n: turnstile.signal()
6     mutex.signal()
7
8 turnstile.wait()
9 turnstile.signal()
10
11 critical point
12
13 mutex.wait()
14     count -= 1
15     if count == 0: turnstile.wait()
16 mutex.signal()
```

In both cases the check is inside the mutex so that a thread cannot be interrupted after changing the counter and before checking it.

Tragically, this code is still not correct. Remember that this barrier will be inside a loop. So, after executing the last line, each thread will go back to the rendezvous.

Puzzle: Identify and fix the problem.

(doesn’t work!)

Allows thread to drop through second mutex and "lap" other threads
Need 2 turnstiles!

- one to force threads to rendezvous before CS
- one to force threads to rendezvous before the next loop
- each turnstiles “resets” the other one
3.6 Reusable barrier

3.6.4 Reusable barrier hint

As it is currently written, this code allows a precocious thread to pass through the second mutex, then loop around and pass through the first mutex and the turnstile, effectively getting ahead of the other threads by a lap.

To solve this problem we can use two turnstiles.

Listing 3.9: Reusable barrier hint

```python
1  turnstile = Semaphore(0)
2  turnstile2 = Semaphore(1)
3  mutex = Semaphore(1)
```

Hint:
Listing 3.10: Reusable barrier solution

```python
# rendezvous
mutex.wait()
count += 1
if count == n:
    turnstile2.wait()  # lock the second
turnstile.signal()   # unlock the first
mutex.signal()

turnstile.wait()  # first turnstile
turnstile.signal()

# critical point

mutex.wait()
count -= 1
if count == 0:
    turnstile.wait()  # lock the first
turnstile2.signal() # unlock the second
mutex.signal()

turnstile2.wait()  # second turnstile
turnstile2.signal()
```

This solution is sometimes called a two-phase barrier because it forces all the threads to wait twice: once for all the threads to arrive and again for all the threads to execute the critical section. Unfortunately, this solution is typical of most non-trivial synchronization code: it is difficult to be sure that a solution is correct. Often there is a subtle way that a particular path through the program can cause an error. To make matters worse, testing an implementation of a solution is not much help. The error might occur very rarely because the particular path that causes it might require a spectacularly unlucky combination of circumstances. Such errors are almost impossible to reproduce and debug by conventional means. The only alternative is to examine the code carefully and "prove" that it is correct. I put "prove" in quotation marks because I don't mean, necessarily, that you have to write a formal proof (although there are zealots who encourage such lunacy). The kind of proof I have in mind is more informal. We can take advantage of the structure of the code, and the idioms we have developed, to assert, and then demonstrate, a number of intermediate-level claims about the program. For example:
We can simplify this with a `signal` API that takes a parameter that takes a number $n$ of signals $\geq 1$

- increments the semaphore by $n$
- potentially unblocks up to $n$ threads
- equivalent to calling `signal` $n$ times in a loop (may be preempted!)
One nice thing about a turnstile is that it is a versatile component you can use in a variety of solutions. But one drawback is that it forces threads to go through sequentially, which may cause more context switching than necessary.

In the reusable barrier solution, we can simplify the solution if the thread that unlocks the turnstile preloads the turnstile with enough signals to let the right number of threads through.

The syntax I am using here assumes that `signal` can take a parameter that specifies the number of signals. This is a non-standard feature, but it would be easy to implement with a loop. The only thing to keep in mind is that the multiple signals are not atomic; that is, the signaling thread might be interrupted in the loop. But in this case that is not a problem.

When the \( n \)th thread arrives, it preloads the first turnstile with one signal for each thread. When the \( n \)th thread passes the turnstile, it "takes the last token" and leaves the turnstile locked again. The same thing happens at the second turnstile, which is unlocked when the last thread goes through the mutex.

Thanks to Matt Tesch for this solution!
next: classic synchronization problems
I. Producer / Consumer
Assume that producers perform the following operations over and over:

Listing 4.1: Basic producer code

```
1 event = waitForEvent()
2 buffer.add(event)
```

Also, assume that consumers perform the following operations:

Listing 4.2: Basic consumer code

```
1 event = buffer.get()
2 event.process()
```

As specified above, access to the buffer has to be exclusive, but `waitForEvent` and `event.process` can run concurrently.

**Puzzle:** Add synchronization statements to the producer and consumer code to enforce the synchronization constraints.

*note: buffer is finite and non-thread-safe!*
4.1 Producer-consumer problem

4.1.5 Finite buffer producer-consumer hint

Add a second semaphore to keep track of the number of available spaces in the buffer.

Listing 4.10: Finite-buffer producer-consumer initialization

1 mutex = Semaphore(1)
2 items = Semaphore(0)
3 spaces = Semaphore(buffer.size())

When a consumer removes an item it should signal `spaces`. When a producer arrives it should decrement `spaces`, at which point it might block until the next consumer signals.

**Hint:**

- While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer.
- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Assume that producers perform the following operations over and over:

Listing 4.1: Basic producer code

1 event = waitForEvent()
2 buffer.add(event)

Also, assume that consumers perform the following operations:

Listing 4.2: Basic consumer code

1 event = buffer.get()
2 event.process()

As specified above, access to the buffer has to be exclusive, but `waitForEvent` and `event.process` can run concurrently.

**Puzzle:** Add synchronization statements to the producer and consumer code to enforce the synchronization constraints.
4.2 Readers-writers problem

The next classical problem, called the Reader-Writer Problem, pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data.

As in the producer-consumer problem, the solution is asymmetric. Readers and writers execute different code before entering the critical section. The synchronization constraints are:

1. Any number of readers can be in the critical section simultaneously.
2. Writers must have exclusive access to the critical section.

In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

### Listing 4.11: Finite buffer consumer solution

```java
1   items.wait()
2   mutex.wait()
3       event = buffer.get()
4   mutex.signal()
5   spaces.signal()
6   event.process()
```

### Listing 4.12: Finite buffer producer solution

```java
1   event = waitForEvent()
2
3   spaces.wait()
4   mutex.wait()
5       buffer.add(event)
6   mutex.signal()
7   items.signal()
```
II. Readers/Writers
story: only one writer in its CS at a time; unlimited number of readers in their CSes simultaneously; writers and readers must access CS separately

i.e., *categorical* mutex
4.2 Readers-writers problem

4.2.1 Readers-writers hint

Here is a set of variables that is sufficient to solve the problem.

Listing 4.13: Readers-writers initialization

1 int readers = 0
2 mutex = Semaphore(1)
3 roomEmpty = Semaphore(1)

Hint:
4.2 Readers-writers problem

4.2.2 Readers-writers solution

The code for writers is simple. If the critical section is empty, a writer may enter, but entering has the effect of excluding all other threads:

Listing 4.14: Writers solution

1. roomEmpty.wait()
2. critical section for writers
3. roomEmpty.signal()

When the writer exits, can it be sure that the room is now empty? Yes, because it knows that no other thread can have entered while it was there.

The code for readers is similar to the barrier code we saw in the previous section. We keep track of the number of readers in the room so that we can give a special assignment to the first to arrive and the last to leave.

The first reader that arrives has to wait for roomEmpty. If the room is empty, then the reader proceeds and, at the same time, bars writers. Subsequent readers can still enter because none of them will try to wait on roomEmpty.

If a reader arrives while there is a writer in the room, it waits on roomEmpty. Since it holds the mutex, any subsequent readers queue on mutex.

Listing 4.15: Readers solution

1. mutex.wait()
2. readers += 1
3. if readers == 1:
   4. roomEmpty.wait() # first in locks
   5. mutex.signal()
4. #critical section for readers
5. 8
6. 9
7. mutex.wait()
8. readers -= 1
9. if readers == 0:
   10. roomEmpty.signal() # last out unlocks
   11. mutex.signal()

The code after the critical section is similar. The last reader to leave the room turns out the lights—that is, it signals roomEmpty, possibly allowing a waiting writer to enter.

Again, to demonstrate that this code is correct, it is useful to assert and demonstrate a number of claims about how the program must behave. Can you convince yourself that the following are true?

- Only one reader can queue waiting for roomEmpty, but several writers might be queued.
- When a reader signals roomEmpty the room must be empty.
4.2 Readers-writers problem

4.2.2 Readers-writers solution

The code for writers is simple. If the critical section is empty, a writer may enter, but entering has the effect of excluding all other threads:

```
Listing 4.14: Writers solution
1:    roomEmpty.wait()
2:    critical section for writers
3:    roomEmpty.signal()
```

When the writer exits, can it be sure that the room is now empty? Yes, because it knows that no other thread can have entered while it was there.

The code for readers is similar to the barrier code we saw in the previous section. We keep track of the number of readers in the room so that we can give a special assignment to the first to arrive and the last to leave.

```
Listing 4.15: Readers solution
1:    mutex.wait()
2:    readers += 1
3:    if readers == 1:
4:        roomEmpty.wait()  # first in locks
5:        mutex.signal()
6:    # critical section for readers
7:    mutex.wait()
8:    readers -= 1
9:    if readers == 0:
10:       roomEmpty.signal()  # last out unlocks
11:       mutex.signal()
```

The code after the critical section is similar. The last reader to leave the room turns out the lights—that is, it signals `roomEmpty`, possibly allowing a waiting writer to enter.

Again, to demonstrate that this code is correct, it is useful to assert and demonstrate a number of claims about how the program must behave. Can you convince yourself that the following are true?

• Only one reader can queue waiting for `roomEmpty`, but several writers might be queued.
• When a reader signals `roomEmpty` the room must be empty.
→ “lightswitch” pattern
Patterns similar to this reader code are common: the first thread into a section locks a semaphore (or queues) and the last one out unlocks it. In fact, it is so common we should give it a name and wrap it up in an object. The name of the pattern is **Lightswitch**, by analogy with the pattern where the first person into a room turns on the light (locks the mutex) and the last one out turns it off (unlocks the mutex). Here is a class definition for a Lightswitch:

```python
class Lightswitch:
    def __init__(self):
        self.counter = 0
        self.mutex = Semaphore(1)

    def lock(self, semaphore):
        self.mutex.wait()
        self.counter += 1
        if self.counter == 1:
            semaphore.wait()
        self.mutex.signal()

    def unlock(self, semaphore):
        self.mutex.wait()
        self.counter -= 1
        if self.counter == 0:
            semaphore.signal()
        self.mutex.signal()
```

`lock` takes one parameter, a semaphore that it will check and possibly hold. If the semaphore is locked, the calling thread blocks on `semaphore` and all subsequent threads block on `self.mutex`. When the semaphore is unlocked, the first waiting thread locks it again and all waiting threads proceed. If the semaphore is initially unlocked, the first thread locks it and all subsequent threads proceed. `unlock` has no effect until every thread that called `lock` also calls `unlock`. When the last thread calls `unlock`, it unlocks the semaphore.
4.2 Readers-writers problem

Using these functions, we can rewrite the reader code more simply:

Listing 4.17: Readers-writers initialization

```python
1    readLightswitch = Lightswitch()
2    roomEmpty = Semaphore(1)
```

readLightswitch is a shared Lightswitch object whose counter is initially zero.

Listing 4.18: Readers-writers solution (reader)

```python
1    readLightswitch.lock(roomEmpty)
2    # critical section
3    readLightswitch.unlock(roomEmpty)
```

The code for writers is unchanged.

It would also be possible to store a reference to roomEmpty as an attribute of the Lightswitch, rather than pass it as a parameter to lock and unlock. This alternative would be less error-prone, but I think it improves readability if each invocation of lock and unlocks specifies the semaphore it operates on.

4.2.3 Starvation

In the previous solution, is there any danger of deadlock? In order for a deadlock to occur, it must be possible for a thread to wait on a semaphore while holding another, and thereby prevent itself from being signaled.

In this example, deadlock is not possible, but there is a related problem that is almost as bad: it is possible for a writer to starve. If a writer arrives while there are readers in the critical section, it might wait in queue forever while readers come and go. As long as a new reader arrives before the last of the current readers departs, there will always be at least one reader in the room.

This situation is not a deadlock, because some threads are making progress, but it is not exactly desirable. A program like this might work as long as the load on the system is low, because then there are plenty of opportunities for the writers. But as the load increases the behavior of the system would deteriorate quickly (at least from the point of view of writers).

Puzzle: Extend this solution so that when a writer arrives, the existing readers can finish, but no additional readers may enter.
recall criteria:

1. no starvation
2. bounded waiting

... but *writer can starve!*
need a mechanism for the writer to prevent new readers from getting “around” it (and into the room)
i.e., “single-file” entry
4.2 Readers-writers problem

4.2.4 No-starve readers-writers hint

Here's a hint. You can add a turnstile for the readers and allow writers to lock it.
The writers have to pass through the same turnstile, but they should check the `roomEmpty` semaphore while they are inside the turnstile. If a writer gets stuck in the turnstile it has the effect of forcing the readers to queue at the turnstile. Then when the last reader leaves the critical section, we are guaranteed that at least one writer enters next (before any of the queued readers can proceed).

Listing 4.19: No-starve readers-writers initialization

```java
1 readSwitch = Lightswitch()
2 roomEmpty = Semaphore(1)
3 turnstile = Semaphore(1)
```

Hint:
4.2 Readers-writers problem

4.2.5 No-starve readers-writers solution

Here is the writer code:

Listing 4.20: No-starve writer solution

1  turnstile.wait()
2     roomEmpty.wait()
3     # critical section for writers
4  turnstile.signal()
5
6  roomEmpty.signal()

If a writer arrives while there are readers in the room, it will block at Line 2, which means that the turnstile will be locked. This will bar readers from entering while a writer is queued. Here is the reader code:

Listing 4.21: No-starve reader solution

1  turnstile.wait()
2  turnstile.signal()
3
4  readSwitch.lock(roomEmpty)
5     # critical section for readers
6  readSwitch.unlock(roomEmpty)

When the last reader leaves, it signals roomEmpty, unblocking the waiting writer. The writer immediately enters its critical section, since none of the waiting readers can pass the turnstile.

When the writer exits it signals turnstile, which unblocks a waiting thread, which could be a reader or a writer. Thus, this solution guarantees that at least one writer gets to proceed, but it is still possible for a reader to enter while there are writers queued.

Depending on the application, it might be a good idea to give more priority to writers. For example, if writers are making time-critical updates to a data structure, it is best to minimize the number of readers that see the old data before the writer has a chance to proceed.

In general, though, it is up to the scheduler, not the programmer, to choose which waiting thread to unblock. Some schedulers use a first-in-first-out queue, which means that threads are unblocked in the same order they queued. Other schedulers choose at random, or according to a priority scheme based on the properties of the waiting threads.

If your programming environment makes it possible to give some threads priority over others, then that is a simple way to address this issue. If not, you will have to find another way.

Puzzle: Write a solution to the readers-writers problem that gives priority to writers. That is, once a writer arrives, no readers should be allowed to enter until all writers have left the system.
exercise for the reader: *writer priority*?
bounded waiting?

- simple if we assume that threads blocking on a semaphore are queued (FIFO)
- i.e., thread blocking longest is woken next
- but semaphore semantics don’t require this
→ *FIFO queue* pattern

goal: use semaphores to build a thread-safe FIFO wait queue

given: non-thread-safe queue
approach:

- protect queue with shared mutex
- each thread enqueues its own *thread-local* semaphores and blocks on it
- to signal, dequeue & unblock a semaphore
class FifoSem:
    def __init__(self, val):
        self.val = val  # FifoSem’s semaphore value
        self.mutex = Semaphore(1)  # possibly non-FIFO semaphore
        self.queue = Queue()  # non-thread-safe queue
    
    def wait(self):
        barrier = Semaphore(0)  # thread-local semaphore
        block = False
        self.mutex.wait()  # modify val & queue in mutex
        self.val -= 1
        if self.val < 0:
            self.queue.enqueue(barrier)
            block = True
        self.mutex.signal()
        if block:
            barrier.wait()  # block outside mutex!
    
    def signal(self):
        self.mutex.wait()  # modify val & queue in mutex
        self.val += 1
        if self.queue:
            barrier = self.queue.dequeue()  # FIFO!
            barrier.signal()
        self.mutex.signal()
henceforth, we will assume that all semaphores have built-in FIFO semantics
III. “Dining Philosophers” problem
typical setup: protect shared resources with semaphores

```python
1 forks = [Semaphore(1) for i in range(5)]
```

```python
1 def left(i): return i
2 def right(i): return (i + 1) % 5
```
solution requirements:

1. each fork held by one philosopher at a time
2. no deadlock
3. no philosopher may starve
4. max concurrency should be possible
Naive solution:

```python
1 def get_forks(i):
2     fork[right(i)].wait()
3     fork[left(i)].wait()
4
5 def put_forks(i):
6     fork[right(i)].signal()
7     fork[left(i)].signal()
```

possible deadlock!
Solution 2: global mutex

```python
def get_forks(i):
    mutex.wait()
    fork[right(i)].wait()
    fork[left(i)].wait()
    mutex.signal()
```

no starvation & max concurrency?

- may prohibit a philosopher from eating when his forks are available
4.4 Dining philosophers

4.4.3 Dining philosophers solution #1

If there are only four philosophers at the table, then in the worst case each one picks up a fork. Even then, there is a fork left on the table, and that fork has two neighbors, each of which is holding another fork. Therefore, either of these neighbors can pick up the remaining fork and eat.

We can control the number of philosophers at the table with a Multiplex named footman that is initialized to 4. Then the solution looks like this:

```python
1 def get_forks(i):
2     footman.wait()
3     fork[right(i)].wait()
4     fork[left(i)].wait()
5
6 def put_forks(i):
7     fork[right(i)].signal()
8     fork[left(i)].signal()
9     footman.signal()
```

In addition to avoiding deadlock, this solution also guarantees that no philosopher starves. Imagine that you are sitting at the table and both of your neighbors are eating. You are blocked waiting for your right fork. Eventually your right neighbor will put it down, because eat can’t run forever. Since you are the only thread waiting for that fork, you will necessarily get it next. By a similar argument, you cannot starve waiting for your left fork. Therefore, the time a philosopher can spend at the table is bounded. That implies that the wait time to get into the room is also bounded, as long as footman has Property 4 (see Section 4.3).

This solution shows that by controlling the number of philosophers, we can avoid deadlock. Another way to avoid deadlock is to change the order in which the philosophers pick up forks. In the original non-solution, the philosophers are “righties”; that is, they pick up the right fork first. But what happens if Philosopher 0 is a leftie?

Puzzle: prove that if there is at least one leftie and at least one rightie, then deadlock is not possible.

Hint: deadlock can only occur when all 5 philosophers are holding one fork and waiting, forever, for the other. Otherwise, one of them could get both forks, eat, and leave.

The proof works by contradiction. First, assume that deadlock is possible. Then choose one of the supposedly deadlocked philosophers. If she’s a leftie, you can prove that the philosophers are all lefties, which is a contradiction. Similarly, if she’s a rightie, you can prove that they are all righties. Either way you get a contradiction; therefore, deadlock is not possible.

Solution 3: limit # diners

```python
footman = Semaphore(4)
```

```python
1 def get_forks(i):
2     footman.wait()
3     fork[right(i)].wait()
4     fork[left(i)].wait()
5
6 def put_forks(i):
7     fork[right(i)].signal()
8     fork[left(i)].signal()
9     footman.signal()
```

no starvation & max concurrency?
Solution 4: leftie(s) vs. rightie(s)

```python
def get_forks(i):
    fork[right(i)].wait()
    fork[left(i)].wait()
```

vs. (at least one of each)

```python
def get_forks(i):
    fork[left(i)].wait()
    fork[right(i)].wait()
```

no starvation & max concurrency?
Solution 4: Tanenbaum’s solution

```python
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)

def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)  # check neighbors' states
    mutex.signal()
    sem[i].wait()  # wait on my own semaphore

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))  # signal neighbors if they can eat
    test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' \
       and state[left(i)] != 'eating' \
       and state[right(i)] != 'eating':
        state[i] = 'eating'
        sem[i].signal()  # this signals me OR a neighbor
```

no starvation & max concurrency?
(let’s mess with this guy)
(starves)
moral: synchronization problems are *insidious*!
IV. Dining Savages

A tribe of savages eats communal dinners from a large pot that can hold $M$ servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.
Chapter 5

Less classical synchronization problems

5.1 The dining savages problem

This problem is from Andrews’s *Concurrent Programming* [1].

A tribe of savages eats communal dinners from a large pot that can hold $M$ servings of stewed missionary. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

Any number of savage threads run the following code:

**Listing 5.1: Unsynchronized savage code**

```python
1 while True:
2   getServingFromPot()
3    eat()
```

And one cook thread runs this code:

**Listing 5.2: Unsynchronized cook code**

```python
1 while True:
2   putServingsInPot(M)
```

rules:
- savages cannot invoke `getServingFromPot` if the pot is empty
- the cook can invoke `putServingsInPot` only if the pot is empty

This problem is based on a cartoonish representation of the history of Western missionaries among hunter-gatherer societies. Some humor is intended by the allusion to the Dining Philosophers problem, but the representation of “savages” here isn’t intended to be any more realistic than the previous representation of philosophers. If you are interested in hunter-gatherer societies, I recommend Jared Diamond’s *Guns, Germs and Steel*[^1], Napoleon Chagnon’s *The Yanomamo*, and Redmond O’Hanlon’s *In Trouble Again*, but not Tierney’s *Darkness in El Dorado*[^2], which I believe is unreliable.

[^1]: Jared Diamond, *Guns, Germs and Steel*.
hint:

```python
servings = 0
mutex    = Semaphore(1)
emptyPot = Semaphore(0)
fullPot  = Semaphore(0)
```

Listing 5.1: Unsynchronized savage code

```
1  while True:
2       getServingFromPot()
3       eat()
```

And one cook thread runs this code:

Listing 5.2: Unsynchronized cook code

```
1  while True:
2       putServingsInPot(M)
```
5.1 The dining savages problem

5.1.2 Dining Savages solution

My solution is a combination of the scoreboard pattern with a rendezvous. Here is the code for the cook:

```
Listing 5.4: Dining Savages solution (cook)
1 while True:
  2   emptyPot.wait()
  3   putServingsInPot(M)
  4   fullPot.signal()
```

The code for the savages is only a little more complicated. As each savage passes through the mutex, he checks the pot. If it is empty, he signals the cook and waits. Otherwise, he decrements \textit{servings} and gets a serving from the pot.

```
Listing 5.5: Dining Savages solution (savage)
1 while True:
  2   mutex.wait()
  3     if servings == 0:
  4       emptyPot.signal()
  5       fullPot.wait()
  6       servings = M
  7       servings -= 1
  8       getServingFromPot()
  9   mutex.signal()
 10   eat()
```

It might seem odd that the savage, rather than the cook, sets \textit{servings} = \textit{M}. That's not really necessary; when the cook runs \textit{putServingsInPot}, we know that the savage that holds the mutex is waiting on \textit{fullPot}. So the cook could access \textit{servings} safely. But in this case, I decided to let the savage do it so that it is clear from looking at the code that all accesses to \textit{servings} are inside the mutex.

This solution is deadlock-free. The only opportunity for deadlock comes when the savage that holds \textit{mutex} waits for \textit{fullPot}. While he is waiting, other savages are queued on \textit{mutex}. But eventually the cook will run and signal \textit{fullPot}, which allows the waiting savage to resume and release the mutex.

Does this solution assume that the pot is thread-safe, or does it guarantee that \textit{putServingsInPot} and \textit{getServingFromPot} are executed exclusively?
shared **servings** counter $\rightarrow$ **scoreboard** pattern

- arriving threads check value of scoreboard to determine system state

- note: scoreboard may consist of more than one variable
V. Baboon Crossing
guarantee rope mutex
max of 5 at a time
no starvation
solution consists of east & west baboon threads:

1. categorical mutex
2. max of 5 on rope
3. no starvation
unsynchronized baboon code (identical for both sides)

```python
1  while True:
2    crossChasm()
```

*hint:*

```python
mulitplex = Semaphore(5)
turnstile = Semaphore(1)
rope = Semaphore(1)
e_switch = Lightswitch()
w_switch = Lightswitch()
```
Patterns similar to this reader code are common: the first thread into a section locks a semaphore (or queues) and the last one out unlocks it. In fact, it is so common we should give it a name and wrap it up in an object. The name of the pattern is **Lightswitch**, by analogy with the pattern where the first person into a room turns on the light (locks the mutex) and the last one out turns it off (unlocks the mutex). Here is a class definition for a Lightswitch:

```python
class Lightswitch:
    def __init__(self):
        self.counter = 0
        self.mutex = Semaphore(1)

    def lock(self, semaphore):
        self.mutex.wait()
        self.counter += 1
        if self.counter == 1:
            semaphore.wait()
        self.mutex.signal()

    def unlock(self, semaphore):
        self.mutex.wait()
        self.counter -= 1
        if self.counter == 0:
            semaphore.signal()
        self.mutex.signal()
```

**Reminder: Lightswitch ADT**

- **lock** takes one parameter, a semaphore that it will check and possibly hold. If the semaphore is locked, the calling thread blocks on `semaphore` and all subsequent threads block on `self.mutex` when the semaphore is unlocked, the first waiting thread locks it again and all waiting threads proceed.
- **unlock** has no effect until every thread that called **lock** also calls **unlock**. When the last thread calls **unlock**, it unlocks the semaphore.

Reminder: Lightswitch ADT
multiplex = Semaphore(5)
turnstile = Semaphore(1)
rope = Semaphore(1)
e_switch = Lightswitch()
w_switch = Lightswitch()

# east side
while True:
    turnstile.wait()
    e_switch.lock(rope)
    turnstile.signal()
    multiplex.wait()
    crossChasm()
    multiplex.signal()
    w_switch.unlock(rope)

# west side
while True:
    turnstile.wait()
    w_switch.lock(rope)
    turnstile.signal()
    multiplex.wait()
    crossChasm()
    multiplex.signal()
    w_switch.unlock(rope)
multiplex = Semaphore(5)
turnstile = Semaphore(1)
rope = Semaphore(1)
mutex_east = Semaphore(1)
mutex_west = Semaphore(1)
east_count = west_count = 0

# west side
while True:
    turnstile.wait()
    mutex_west.wait()
    west_count++
    if west_count == 1:
        rope.wait()
    mutex_west.signal()
    turnstile.signal()
    multiplex.wait()
    crossChasm()
    multiplex.signal()

    mutex_west.wait()
    west_count--
    if west_count == 0:
        rope.signal()
    mutex_west.signal()

# east side
while True:
    turnstile.wait()
    mutex_east.wait()
    east_count++
    if east_count == 1:
        rope.wait()
    mutex_east.signal()
    turnstile.signal()
    multiplex.wait()
    crossChasm()
    multiplex.signal()

    mutex_east.wait()
    east_count--
    if east_count == 0:
        rope.signal()
    mutex_east.signal()
… many, many more contrived problems await you in the little book of semaphores!