Concurrency, Races, & Synchronization

CS 450: Operating Systems
Sean Wallace <swallac6@iit.edu>
Agenda

• Concurrency: what, why, how
  • Concurrency-related problems
• Locks & Locking strategies
• Concurrent programming with semaphores
Concurrency: what, why, how
Concurrency (in computing) = 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

• That is, computations run *at the same time*

• …But, concurrency can also be achieved with *single-CPU multiplexing*

• That is, via *context switches*
concurrency
But, 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 blob = 0;

int main(int argc, const char * argv[]) {
    pid_t pid;
    for (int i = 0; i < 5; i++) {
        if ((pid = fork()) == 0) {
            blob += 1;
            printf("Child %d blob = %d\n", i, blob);
            exit(0);
        }
    }
    printf("Parent created child %d\n", pid);
}

return 0;
}
```

Parent created child 12678
Parent created child 12679
Child 0 blob = 1
Parent created child 12680
Child 1 blob = 1
Child 2 blob = 1
Child 3 blob = 1
Parent created child 12681
Parent created child 12682
Child 4 blob = 1
Program ended with exit code: 0
• 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*; that is:

  • One path through program

  • One stack

  • Blocking this thread (e.g., with I/O) blocks the entire process
But, the single-thread-per-process model is not always ideal nor 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(); // block 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
#define DIM 5

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)
But, *who* 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 discussion.
• 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 thread, 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 (coming later)
• 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")
/* Creates a new thread of execution. */
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void *),
        void *arg);

/* Causes the calling thread to wait for the termination
   of the specified thread; thread "reaping" */
int pthread_join(pthread_t thread,
        void **value_ptr);

/* Terminates the calling thread. */
void pthread_exit(void *value_ptr);
```c
int blob = 0;

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

int main(int argc, const char * argv[]) {
    pthread_t tid;
    for (int i = 0; i < 5; i++) {
        pthread_create(&tid, NULL, inc_blob, (void *)i);
        printf("Created thread %ld\n", (long)tid);
    }
    pthread_exit(NULL);
    return 0;
}
```

Run 1:

- Thread 0 blob = 10000
- Created thread 4297592832
- Thread 1 blob = 20000
- Created thread 4297592832
- Thread 2 blob = 30000
- Created thread 4297592832
- Thread 3 blob = 40000
- Created thread 4297592832
- Thread 4 blob = 50000
- Created thread 4297592832
- Program ended with exit code: 0

Run 2:

- (?!)
int blob = 0;

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

int main(int argc, const char * argv[]) {
    pthread_t tid;
    for (int i = 0; i < 5; i++) {
        pthread_create(&tid,
            NULL,
            inc_blob,
            (void *)i);
        pthread_join(tid, NULL);
        printf("Created thread %ld\n", (long)tid);
    }
    pthread_exit(NULL);
    return 0;
}
• Note: pthreads API doesn’t specific whether implementation is kernel/user

• Platform dependent

• Most modern Unixes provided kernel-level threading support
Sample fiber library: libtask (swtch.com/libtask/)
int blob = 0;

void *inc_task(void *num) {
    for (int i = 0; i < 3; i++) {
        for (int i = 0; i < 10000; i++) {
            blob += 1;
        }
        printf("Thread %d blob = %d\n", (int)num, blob);
        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 */
    }
}
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(2, "out of memory\n");
            abort();
        }
    }
    t->alltaskslot = nalltask;
    alltask[nalltask++] = t;
    taskready(t);
    return id;
}

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(2, "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);
    sigemptyset(&zero);
    sigprocmask(SIG_BLOCK, &zero, &t->context.uc.uc_sigmask);

    /* must initialize with current context */
    if(getcontext(&t->context.uc) < 0){
        fprintf(2, "getcontext: %r\n");
        abort();
    }

    return t;
}
• \texttt{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 \texttt{setjmp/longjmp})
static void
ccontextswitch(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;        /* machine state */
    ...
};

#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

.globl GET
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)
#define DIM 50

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];
            }
        }
    }
}

Run time, with DIM=50, 500 iterations:

real 0m0.313s
user 0m0.310s
sys 0m0.002s
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 (int i = 0; i < DIM; i++) {
        for (int 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 1m6.973s
user 0m6.320s
sys 1m36.200s
Single 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 *independent* (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: \( S = \frac{1}{\frac{P}{N} + (1 - P)} \)

\( \dagger P \rightarrow 1; \ S \rightarrow N \)
\( \ddagger N \rightarrow \infty; \ S \rightarrow \frac{1}{1 - P} \)
source: https://upload.wikimedia.org/wikipedia/commons/e/ea/AmdahlsLaw.svg
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 number of cores
• **Gustafson’s Law** computes speedup based on:
  
  • N cores
  
  • *non*-parallelizable fraction, P
Speedup: \[ S = N - P \cdot (N - 1) \]

\[ \dagger P \rightarrow 1; \ S \rightarrow 1 \]
\[ \ddagger P \rightarrow 0; \ S \rightarrow N \]

Predicted speedup is *linear* with respect to number of cores!
Number of Cores: $N$

- $x - 0.1 \times (x-1)$
- $x - 0.2 \times (x-1)$
- $x - 0.3 \times (x-1)$
- $x - 0.4 \times (x-1)$
- $x - 0.5 \times (x-1)$
- $x - 0.6 \times (x-1)$
- $x - 0.7 \times (x-1)$
- $x - 0.8 \times (x-1)$
- $x - 0.9 \times (x-1)$
• Amdahl's Law approximately suggests:

  • Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph.

• Gustafson's Law approximately states:

  • Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, if the car spent one hour at 30 mph, it could achieve this by driving at 120 mph for two additional hours, or at 150 mph for an hour, and so on.
• Amdahl’s vs. Gustafson’s:
  • Latter has rosier implication for big data analysis / data science
    • But not all datasets naturally expand / increase in resolution
  • Both street the importance of maximizing the parallelizable fraction
Some of the primary challenges of concurrent programming are to:

1. Identity thread interdependencies
2. Identify (1)’s potential ramifications
3. Ensure correctness
E.g., final change in count? (expected = 2)

Thread A
```
> a1 count = count + 1
```

Thread B
```
> b1 count = count + 1
```

Interdependency: shared var count
Factoring in machine-level granularity:

Thread A

a1 lw (count), %r0
a2 add $1, %r
a3 sw %r0, (count)

Thread B

b1 lw (count), %r0
b2 add $1, %r0
b3 sw %r0, (count)

Answer: either +1 or +2
Race condition(s) exist when results are dependent on the order of execution or concurrent tasks.
Shared resource(s) are the problem

or, more specifically, *concurrent mutability* of those 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 application logic from synchronization details

• Another instance of policy vs. mechanism

• This can be hard to get right!
Most common technique for implementing synchronization is via software “locks”

- explicitly required & released by consumer 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

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

Thread B

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

Diagram:
- Thread A (`T_A`)
- Thread B (`T_B`)
- Lock
- `acquire`
- `count`
Thread A

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

Thread B

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

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

Thread B

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

```java
a1 count = count + 1
```

Thread B

```java
b1 count = count + 1
```

![Diagram showing thread interactions](image-url)
Thread A

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

Thread B

\[ b1 \text{ count} = \text{count} + 1 \]
• Locking can be:
  
  • **Global** (*coarse-grained*)
  
  • **Per-resource** (*fine-grained*)
Coarse-grained locking policy
Coarse-grained locking policy

count  buff  logfile  GUI

T_A  T_B  T_C  T_D
Coarse-grained locking policy

```plaintext
count  buff  logfile  GUI
```

```
TA  TB  TC  TD
```
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, we 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 video

Watch an introduction to semaphores (and Free Books) I presented at Northeastern University:

Model of execution

- On some machines, x++ is atomic.
- But let's not count on it.
- Assume:
  - Result of concurrent writes is undef.
  - Result of concurrent read-write is undef.
  - Concurrent reads are ok.
- Threads can be interrupted at any time.
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.
Initialization syntax:

```python
fred = Semaphore(1)
```
Operation names?

fred.increment_and_wake_a_waiting_process_if_any()
fred.decrement_and_block_if_the_result_is_negative()

fred.increment()
fred.decrement()

fred.signal()
fred.wait()

fred.V()
fred.P()
Operation names?

```plaintext
fred.increment_and_wake_a_waiting_process_if_any()
fred.decrement_and_block_if_the_result_is_negative()

fred.increment()
fred.decrement()

fred.signal()
fred.wait()

fred.V()
fred.P()
```
• How to use semaphores for synchronization?

1. Identity essential usage “patterns”

2. Solve “classic” synchronization problems
Essential synchronization criteria:

1. Avoid *starvation*

2. Guarantee *bounded waiting*

3. No assumptions *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. Rendevous

Thread A

statement a1
statement a2

Thread B

statement b1
statement b2

Guarantee: $a_1 < b_2$, $b_1 < a_2$
aArrived = Semaphore(0)
bArrived = Semaphore(0)

Thread A
statement a1
aArrived.signal()
bArrived.wait()
statement a2

Thread B
statement b1
bArrived.signal()
aArrived.wait()
statement b2
Note: Swapping wait/signal → Deadlock!

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement a1</td>
<td>statement b1</td>
</tr>
<tr>
<td>bArrived.wait()</td>
<td>aArrived.wait()</td>
</tr>
<tr>
<td>aArrived.signal()</td>
<td>bArrived.signal()</td>
</tr>
<tr>
<td>statement a2</td>
<td>statement b2</td>
</tr>
</tbody>
</table>
II. Mutual Exclusion

Thread A

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

Thread B

\[ \text{count} = \text{count} + 1 \]
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()
III. \texttt{multiplex} = \texttt{Semaphore(N)}

\begin{verbatim}
multiplex.wait()
critical section
multiplex.signal()
\end{verbatim}
IV. Generalized Rendezvous / Barrier

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

Listing 3.2: Barrier code

rendezvous
critical point
n = the number of threads
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
rendevous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: barrier.signal()

barrier.wait()
barrier.signal()

critical point
rendevous

mutex.wait()
    count = count + 1
mutex.signal()

if count == n: turnstile.signal()

turnstile.wait()
turnstile.signal()

critical point

State of turnstile after all threads make it to critical point?
rendevous

mutex.wait()
    count = count + 1
    if count == n: turnstile.signal()
mutex.signal()

turnstile.wait()
turnstile.signal()

critical point

Fix for non-determinism (but still off by one)
Next: would like a **reusable** barrier

Need to **re-lock** turnstile
rendevous

mutex.wait()  
  count = count + 1 
  if count == n: turnstile.signal() 
mutex.signal() 

turnstile.wait() 
turnstile.signal() 

critical point 

mutex.wait()  
  count = count - 1 
  if count == 0: turnstile.wait()  
mutex.signal() 

(Doesn’t work!)
#rendevous

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

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

mutex.wait()
    count = count + 1
    if count == n:
        turnstile.signal(n)  # unlock the first
mutex.signal()

turnstile.wait()  # first turnstile

#critical point

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

turnstile2.signal()  # second turnstile
Next: classic synchronization problems
I. Producer / Consumer
Assume that producers perform the following operations over and over:

Listing 4.1: Basic producer code

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

Also, assume that consumers perform the following operations:

Listing 4.2: Basic consumer code

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

Important: buffer is finite and non-thread-safe!
- finite, non-thread-safe buffer
- 1 semaphore per item/space

```python
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(buffer.size())
```
Listing 4.11: Finite buffer consumer solution

```java
items.wait()
mutex.wait()
    event = buffer.get()
mutex.signal()
spaces.signal()
event.process()
```

Listing 4.12: Finite buffer producer solution

```java
event = waitForEvent()
spaces.wait()
mutex.wait()
    buffer.add(event)
mutex.signal()
items.signal()
```
II. Readers/Writers
Categorical mutex
Listing 4.13: Readers-writers initialization

```c
int readers = 0
mutex = Semaphore(1)
roomEmpty = Semaphore(1)
```
Listing 4.14: Writers solution

```java
roomEmpty.wait()
critical section for writers
roomEmpty.signal()
```
Listing 4.14: Readers solution

```python
mutex.wait()
    readers += 1
    if readers == 1:
        roomEmpty.wait()    # first in locks
mutex.signal()

# critical section for readers

mutex.wait()
    readers -= 1
    if readers == 0:
        roomEmpty.signal()  # last out unlocks
mutex.signal()
```
→ “lightswitch” pattern
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()
readLightswitch is a shared Lightswitch object whose counter is initially zero.

Listing 4.18: Readers-writers solution (reader)

```python
readLightswitch.lock(roomEmpty)
# critical section
readLightswitch.unlock(roomEmpty)
```
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
Listing 4.19: No-starve readers-writers initialization

```python
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)
turnstile = Semaphore(1)
```
Listing 4.20: No-starve writer solution

```java
turnstile.wait()
   roomEmpty.wait()
   // critical section for writers
turnstile.signal()

roomEmpty.signal()
```

Listing 4.21: No-starve reader solution

```java
turnstile.wait()
turnstile.signal()

readSwitch.lock(roomEmpty)
   // critical section for readers
readSwitch.unlock(roomEmpty)
```
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
→ FIFIO 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  = dequeue()      # 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.append(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.popleft() # FIFO!
            barrier.signal()
        self.mutex.signal()
From here on out, we will assume that all semaphores have *built-in* FIFO semantics.
III. “Dining Philosophers” problem
Typical setup: protect shared resources with semaphores

Listing 4.30: Variables for dining philosophers

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

Listing 4.29: Which fork?

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

def put_forks(i):
    fork[right(i)].signal()
    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
Solution 3: limit # of diners

```python
footman = Semaphore(4)

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

def put_forks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
    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 of the story: synchronization problems are *insidious*!
IV. Dining Savages

“I don’t know... it tastes funny!”
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.

Listing 5.1: Unsynchronized savage code

```python
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

Listing 5.2: Unsynchronized cook code

```python
while True:
    putServingsInPot(M)
```
Listing 5.1: Unsynchronized savage code

```python
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

**Listing 5.2: Unsynchronized cook code**

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

**Rules:**

- Savages cannot invoke `getServingFromPot` if the pot is empty
- The cook can invoke `putServingsInPot` only if the pot is empty
while True:
    getServingFromPot()
    eat()

And one cook thread runs this code:

while True:
    putServingsInPot(M)


Listing 5.1: Unsynchronized savage code

Listing 5.2: Unsynchronized cook code

hint:

servings = 0
mutex = Semaphore(1)
emptyPot = Semaphore(0)
fullPot = Semaphore(0)
while True:
    emptyPot.wait()
    putServingsInPot(M)
    fullPot.signal()

Listing 5.5: Dining Savages solution (savage)

while True:
    mutex.wait()
    if servings == 0:
        emptyPot.signal()
        fullPot.wait()
        servings = M
    servings -= 1
    getServingFromPot()
    mutex.signal()

eat()
Shared servings counter → 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:
  
  • Categorial mutex
  
  • Max of 5 on rope
  
  • No starvation
while True:
    climbOnRope()
    CrossChasm()

**hint:**

```
multiplex = Semaphore(5)
turnstile = Semaphore(1)
rope      = Semaphore(1)
e_switch  = Lightswitch()
w_switch  = Lightswitch()
```
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()
```python
multiplex = Semaphore(5)
turnstile = Semaphore(1)
rope = Semaphore(1)
e_switch = Lightswitch()
w_switch = Lightswitch()

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

while True:
    # east side
    turnstile.wait()
    e_switch.lock(rope)
    turnstile.signal()
    multiplex.wait()
    climbOnRope()
    crossChasm()
    multiplex.signal()
    e_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
turnstile.wait()
mutex_west.wait()
    west_count++
    if west_count == 1:
        rope.wait()
    mutex_west.signal()
turnstile.signal()

multiplex.wait()
    # cross the chasm
multiplex.signal()

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

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

multiplex.wait()
    # cross the chasm
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!