COM S 352 Midterm Exam Study Guide Fall 2023

List of terms from reading/lecture

- Processes
  - Program
  - Process
  - Address space
  - Program counter
  - CPU registers
  - Stack pointer
  - PCB (Process control block)
  - Scheduler
  - Running (state)
  - Ready (state)
  - Blocked (state)
  - Context Switch
  - Parent (process)
  - Child (process)
  - Zombie (process)
  - Orphan (process)
  - Process System calls
    - fork()
    - exec()
    - wait()
    - pipe()
    - dup()
  - Pipes
  - Time-sharing
  - Multiprogramming
  - Multitasking
  - User mode (CPU)
  - Kernel mode (CPU)
  - Privileged instruction
  - System call
  - Trap
- Scheduling
  - Job (or CPU burst)
  - I/O burst
  - CPU Bound
  - I/O Bound
  - Preemption

    - Turnaround time
    - Response time
    - Scheduling policies
        - FIFO
        - SJF
        - STCF
        - RR
        - Lottery
        - Stride
        - MLFQ
        - CFS
    - Time-slice (quanta in RR)
    - Oracle (requirement to see into the future)
    - Tickets (Lottery)
    - Starvation
    - Priority boost (MLFQ)
    - Virtual runtime (CFS)
    - Nice value (CFS)
- Memory
    - Address space (virtual)
    - Physical memory
    - Program code
    - Heap
    - Stack
    - Memory virtualization
    - Virtual address
    - Physical address
    - Address translation
    - Base and Bound
    - MMU (Memory Management Unit)
    - Segmentation
    - Segment (address space)
    - Segmentation fault
    - Sparse address space
    - Extern fragmentation
    - Internal fragmentation
    - Compaction
    - Best fit (free-space management policy)
    - Paging (memory)
    - Page table
    - Page table entry
    - Page
    - Frame
    - VPN (virtual page number)

    - PFN (physical frame number)
    - Offset (from start of page)
    - Valid bit (page table)
    - TLB (Translation-Lookaside Buffer)
    - TLB hit
    - TLB miss
    - TLB cache entry
    - Valid bit (TLB cache entry)
    - Spatial locality
    - Temporal locality
    - Memory hierarchy
    - Swap space
    - Block (swap on disk)
    - Present bit (swap)
    - Cache hit
    - Cache miss
    - Page fault
    - Page out
    - Page in
    - High watermark
    - Low watermark
    - Reference string
    - Cold-start miss
    - Replacement Policies
        - OPT (optimal)
        - FIFO (first in first out)
        - Random
        - LRU (least-recently used)
        - Clock Algorithm
    - Belady's Anomaly
    - Dirty bit (swap)
    - Thrashing
- Concurrency
    - Threads
    - TCB (Thread Control Block)
    - Concurrency vs Parallelism
    - Race condition
    - POSIX
    - pthreads
        - pthread_create()
        - pthread_exit()
        - pthread_join()
    - Mutex
        - pthread_mutex_lock()

        - pthread_mutex_unlock()
    - Mutual exclusion
    - test-and-set
    - Spin lock
    - Spinning loops
    - Condition variables
        - pthread_cond_wait()
        - pthread_cond_signal()
    - Producer/consumer (bounded buffer) problem
    - Semaphore
        - sem_init()
        - sem_wait()
        - sem_post()
    - Binary semaphore
    - Reader-Writers problem

Module 1

1. For each state transition below, state whether there is a context switch involved. If there is a context switch, describe the processes involved. How are they chosen?

a. ready -> running
b. running -> blocked
c. blocked -> ready

a. There is a context switch. The scheduler replaces the currently running process and replaces it with the next (based on some algorithm) process from the ready queue.
b. There is a context switch. The currently running process is placed into a blocked state, the scheduler replaces it with the next (based on some algorithm) process from the ready queue.
c. There is no context switch. When a process because ready to run it is placed in the ready queue, assuming there is no preemption, the currently running process remains running.

2. Three programs are serviced in a multiprogramming system. Program A contains 50ms of computation followed by 100ms of I/O on hardware device 1. Program B contains 20ms of computation followed by 50ms of I/O on hardware device 2. Program C contains 50ms of computation followed by 100ms I/O on hardware device 2. Each device can service only one I/O request at a time. What is the minimum time it will take to complete all three programs? Create a table like Figure 4.4 in the reading to show the operation of the three programs.

Time	A	B	C	Notes
0	Ready	Ready	Running	
50	Running	Ready	Blocked	C waiting 100ms for Device 2 I/O
100	Blocked	Running	Blocked	A waiting 100 ms for Device 1 I/O
120	Blocked	Blocked	Blocked	B waiting 50 ms for Device 2 I/O
150	Blocked	Blocked	-	C is finished
170	Blocked	-	-	B is finished
200	-	-	-	A is finished

All programs finished at 200.

3. Describe the purpose of each of the following terms individually:

- system call
- interrupt
- trap

Now, describe how the three are related to each other.

System Call – call a system library and transfer control to the OS in kernel mode
Interrupt – signal from a hardware source that gives CPU control to the interrupt handler (owned by the OS)
Trap – signal from a software source that gives CPU control to the interrupt handler (owned by the OS)

System calls need to execute in kernel mode, kernel mode can be entered by an interrupt or trap. Software initiates a system call by causing a trap. Hardware (e.g., a device driver or timer) changes control to the OS by an interrupt.

4. The RISC-V architecture that we will use in examples throughout this class allow processors to be implemented with three privilege levels defined as follows (http://docs.keystone-enclave.org/en/latest/Getting-Started/How-Keystone-Works/RISC-V-Background.html):

> *Privilege level defines what the running software can do during its execution. Common usage of each privilege level is as follows:*
>
> - *U-mode: user processes*
> - *S-mode: kernel (including kernel modules and device drivers), hypervisor*
> - *M-mode: bootloader, firmware*

However, the specification allows simple embedded processors to implement just M-mode (i.e., all code on these simple processor implementations must run at the same privilege level).

What is the purpose of multiple privilege levels? Compare and contrast multiple privilege levels vs a single privilege level, name at least one advantage for each.

We don't want the user processes to be able to access each other's memory or the memory of the OS or other resources that they should not have privilege to access. When a process executes in user mode it is restricted it what instructions it can execute. Only OS code executes in kernel mode where there are generally no restrictions. The RISC-V architecture has an additional mode that can execute instructions that are only required during bootloading.

Advantage of single level: With a single-level, OS libraries can be called with simple function calls. System calls are more expensive because they require a trap and context switch.

Advantage of multiple levels: We can restrict what user processes are able to do to protect memory and resources.

5. The following application code demonstrates what is called a "fork bomb". What negative consequence might this application have on the system? What might an OS do to prevent it?

```
/* WARNING: DO NOT EXECUTE THIS CODE!       */
/* SSG MAY GIVE YOU A STERN WARNING OR WORSE */
int main(int args, char *argv[]) {
   int rc = 0;
   while(rc == 0) {
      rc = fork();
   }
   wait(NULL);
   return 0;
}
```

The result of the loop is that the child forks a new process and this happens recursively. So an infinite number of processes are created. The parents call wait so they never terminate. Eventually the system will run out of memory or process ids.

To prevent it set a maximum number of processes.

6. A shell is an application that enables users to run and control other applications through a command line interface. Using the POSIX API for processes (e.g., fork, exec, wait, dup and pipe)
list the steps a shell might take to execute the following user commands:

a) Start another application. For example, the user invokes the grep application with two command line arguments like this:
$ grep COMS352 *

1. **Shell calls fork**
2. **Child calls exec with "grep COMS352 *"**

b) Pipe the standard out of one application into the standard in of another application. For example, the user pipes the output of the grep application into the input of the wc application like this:
$ grep COMS352 * | wc –l

Module 2

1. Of these two types of programs:

a. I/O bound (interactive)
b. CPU bound

which is more likely to have voluntary context switches, and which is more likely to have nonvoluntary context switches? Explain your answer.

I/O-bound is voluntary. The program is more likely to wait for I/O to become available before it uses up its time slice. So, it is more likely to voluntarily relinquish the CPU. CPU-bound is non-voluntary. The program is likely to use the CPU for its entire time quantum.

2. OSTEP describes the problem of starvation in section 8.2 and shows how priority boost can be used to avoid it in MLFQ. Consider other scheduler policies. Explain why each of the following policies can or cannot result in starvation?

a. FIFO
b. STCF
c. RR
d. Lottery

(a) starvation – job can have infinite loop and starve other jobs
(b) starvation – many short jobs can prevent longer job from getting run time
(c) no starvation – guaranteed execution of every process within (N * time slice)
(d) no starvation – every lottery selection process has chance of executing based on how many tickets it has

3. CPU efficiency can be defined as:

```
process_running_time / (process_running_time + os_overhead)
```

Between STCF and RR which do think will typically have better CPU efficiency? What are the overheads involved? Explain your answer.

RR has timer preemptions that cause extra context switches that STCF will not have. Context switches contribute to significant overhead of the scheduler. Therefore, RR is less CPU efficient.

4. Consider the following set of jobs, with times given in milliseconds:

Job	Arrival Time	Runtime
A	0	20
B	2	20
C	5	5
D	10	10

a. Draw four Gantt charts that illustrate the execution of these jobs using the following scheduling algorithms: FIFO, SJF, STCF and RR (with quantum = 5). The charts must clearly label the job and the start and end time of each run.

FIFO

```
|   A   |   B   | C |  D  |
0       20      40  45    55
```

SFJ

```
|   A   | C |  D  |   B   |
0       20  25    35      55
```

STCF

```
|A|C|  D  |   A   |   B   |
0 5 10    20      35      55
```

RR

```
| A | B | C | D | A | B | D | A | B | A | B |
0   5   10  15  20  25  30  35  40  45  50  55
```

b. What is the average turnaround time for each of the scheduling policies? Show calculations.

FIFO
((20 – 0) + (40 – 2) + (45 -5) + (55 – 10)) / 4

SFJ
((20 – 0) + (55 – 2) + (25 -5) + (35 – 10)) / 4

STCF
((35 – 0) + (55 – 2) + (10 -5) + (20 – 10)) / 4

RR
((50 – 0) + (55 – 2) + (15 -5) + (35 – 10)) / 4

c. What is the average response time for each of these scheduling policies? Show calculations.

FIFO
((0 – 0) + (20 – 2) + (40 - 5) + (45 – 10)) / 4

SFJ
((0 – 0) + (35 – 2) + (20 - 5) + (25 – 10)) / 4

STCF

((0 – 0) + (35 – 2) + (5 - 5) + (10 – 10)) / 4

RR
((0 – 0) + (5 – 2) + (10 -5) + (15 – 10)) / 4

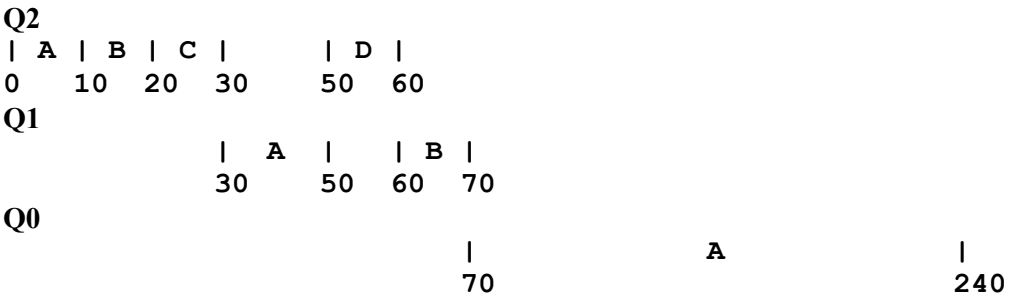
5. Consider the following set of jobs, with times given in milliseconds.

Job	Arrival Time	Runtime
A	0	200
B	5	20
C	10	10
D	50	10

Assume we are using a MLFQ scheduler with 3 priority levels Q2: quanta=10, Q1: quanta=20, Q0: quanta=50.

The MLFQ follows rules 1-4 from OSTEP, it does not have priority boost rule 5.

Draw a Gantt chart that illustrates the execution of the jobs.



6. Assume that two processes, A and B, are running on a Linux system. The nice values of A and B are −5 and +5, respectively. If both A and B have the same positive `vruntime`, and neither has had its `vruntime` reset due to sleeping, which has run the longest actual time on the CPU? Explain your answer.

A is less nice so its vruntime grows more slowly with actual time. Therefore, it will run for more actual time than B which has the same vruntime.

Module 3

1. Suppose an MMU manages memory using base and bounds registers. Assume the base and bounds for a particular process are:
base = 22,000
bounds = 800

A program performs the following loads (reads) and stores (writes) from/to memory. What is the physical memory location of each load or store. If the operation will not be allowed explain why.

a. Load from address 0
b. Load from address 500
c. Store to address 900

virtual_address + base = physical_address

a. 0 + 22,000 = 22,000
b. 500 + 22,000 = 22,500
c. 900 is greater than the bounds which means the process is not allowed to access that memory. The MMU will case a trap so the OS can decide what to do with the process that attempted the illegal memory access.

2. What is the size of addressable memory (maximum number of bytes that can be stored) for each of the following memory address sizes?

a. 16-bit address
b. 24-bit address
c. 32-bit address

a. 2^16 = 65,536 bytes, note that addresses will range from 0 to 65,535.
b. 2^24 = 16,777,216 bytes
c. 2^32 = 4,294,967,296 bytes

3. The following program is run to find the location of code, heap, and stack in virtual memory.

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

int main(int argc, char *argv[]) {
    printf("code  %lu\n", main);
    printf("heap  %lu\n", malloc(1));
    printf("stack %lu\n", &argc);
    return 0;
}
```

The output is:

```
code  5287601
heap  17573467
stack 8731878194284
```

State each of these addresses using binary SI units (e.g., 1KB=2^10) rounded to the nearest whole number. For example, 45,876 is 45,876 / 2^10 = 45KB.

5,287,601 / 2^20 = 5.0MB = 5MB
17,573,467 / 2^20 = 16.8MB = 17MB
8,731,878,194,284 / 2^40 = 7.9TB = 8TB

4. Describe how a process' physical memory can be relocated when using base and bounds. How does it compare to static relocation in memory?

To relocate:

1) **If process is currently running cause a trap to switch out of the process.**
2) **Copy the contiguous block of physical memory to another location.**
3) **Update the base register to be the start address of the new location in physical memory.**

Static relocation of physical process memory requires identifying all pointers in the executing program and updating them to point to the new location in physical memory. Without making assumptions about the program identifying pointers in a running program is extremely difficult if not impossible to do.

5. Suppose an MMU manages memory with per segment base and size registers. Addresses are 16 bits in size. The 2 most significant bits encode the segment. The 14 least significant bits encode the offset. Segments are defined as follows:

Segment Number	**Segment Name**	**Base**	**Size**
0	code	0x0100	0x0100
1	heap	0x0A00	0x0200
2	stack	0x1000	0x0100
3	undefined	-	-

For each of the following virtual addresses state where the data will be stored in physical memory.

a. 0x401A
b. 0x0000
c. 0x8001

a. 0x401A = b0100 0000 0001 1010
segment 1: heap
0x001A + 0x0A00 = 0x0A1A

b. 0x0000 = b0000 0000 0000 0000
segment 0: code
0x0000 + 0x0100 = 0x0100

c. 0x8001 = b1000 0000 0000 0001
segment 2: stack
0x0001 + 0x1000 = 0x1001

Module 4

1. For each of the following memory allocation schemes explain why it does or does not experience external fragmentation or internal fragmentation.

a. Base and bounds

External fragmentation: yes
Internal fragmentation: no
Variable sized allocations can lead to free spaces that are not large enough to fit an entire process.

b. Segmentation

External fragmentation: yes
Internal fragmentation: no
Variable sized allocations can lead to free spaces that are not large enough to fit an entire segment.

c. Paging

External fragmentation: no
Internal fragmentation: yes
Pages and frames are the same size so there will never be wasted external free memory that cannot be allocated to a page. It is possible for the allocated memory to not completely fill a page, therefore, there can be internal fragmentation.

2. Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?

a. Base and bounds

Copy process to new location in physical memory and then update the base and bounds registers.

b. Segmentation

If there is free space after the segment then the segment can be expanded by updating the size register for that segment.
If there is not enough free space after the segment then it must be copied to a new physical location and the base and bounds registers updated.

c. Paging

Allocated free physical frames to unused pages by updating the page table.

a. 22,000

4KB is 4,096
22,000 / 4,096 = 5
22,000 % 4,096 = 1,520

b. 77,056
77,056 / 4,096 = 18
77,056 % 4,096 = 3,328

c. 197,012
197,012 / 4096 = 48
197,012 % 4096 = 404

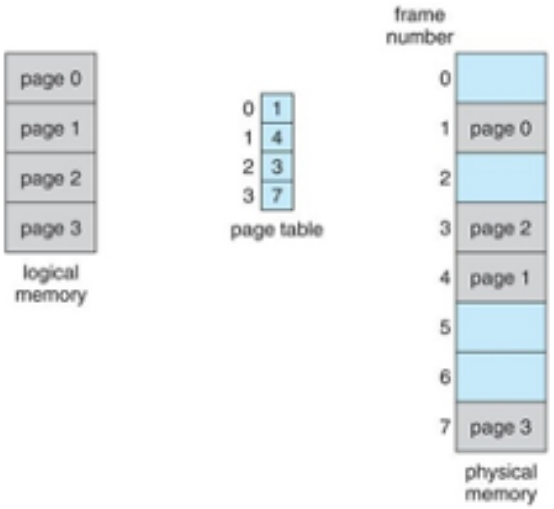
4. Suppose a process' logical address consists of 4 pages and the page size is 2KB. The page table of the process is given in below. Compute the physical address for each of the following logical addresses (provided as decimal numbers).

a. 1,018
Page number is 1,018 / 2,048 = 0
Offset is 1,018

Page 0 maps to frame 1
Frame 1 starts at address 2,048
Physical address is 1,018 + 2,048 = 3,066

b. 6,976
Page number is 6,976 / 2,048 = 3
Offset is 6,976 % 2,048 = 832

Page 3 maps to frame 7
Frame 7 starts at address 7 * 2,048 = 14,336
Physical address is 14,336 + 832 = 15,168



5. On some systems, virtual and physical addresses are different sizes. Consider a small embedded system with a virtual address space of 4,096 pages and a 2KB page size. The physical memory has a maximum capacity of 1,024 frames.

a. How many bits are required for addresses in the virtual address space?
4,096 * 2,048 = 8,388,608 bytes = 2^23 bytes
Therefore 23 bits are required.

b. How many bits are required for addresses in physical memory
1,024 * 2,048 = 2,097,152 = 2^21 bytes
Therefore 21 bits are required.

Module 5

1. Suppose that a machine has 48-bit virtual addresses and 32-bit physical addresses.
   a. If pages are 4KB, how many entries are in the page table if it has only a single level? Explain.

The number of entries in the page table is the same as the number of pages in the virtual address space.

The total size of the virtual address space is 2^48 bytes.

The size of a page is 4KB

Therefore, there are 2^48 / (4*2^10) = ~68.7 billion entries.

   b. Suppose this same system has a TLB with 32 entries. Furthermore, suppose that a program contains instructions that fit into one page and it sequentially reads long integer elements from an array that spans thousands of pages. How effective will the TLB be for this case?

Suppose an integer is 8 bytes long. Then 4*2^10/8 = 512 integers fit on a page. The array spans many pages, on every new page that is accessed there is one cache miss. The array is accessed sequentially, resulting in high spatial locality, which means the page entry is put into cache only once and after it is evicted will never be needed again. With the example numbers there will be a miss rate of about 1/512 = 0.195%.

2. Suppose that a machine has 38-bit virtual addresses and 32-bit physical addresses.
   a. What is the main advantage of a multilevel page table over a single-level one?

A multilevel page table does not require page table entries for every unallocated page. Consider a newly created process that only requires 3 pages (one each for code, heap and stack). The multilevel page table would only need to be a few pages, while a single level page table would require millions of pages.

   b. With a two-level page table, 16-KB pages, and 4-byte entries, from the VPN how many bits should be allocated for the page directory index and how many for the page table index? Explain.

Number of offset bits: log2(16KB) = log2(2^14) = 14 bits

Number of entries per page: 16 * 2^10 / 4 = 4,096 entries

Number of bits required to address 1 page's entries: log2(4,096) = log2(2^12) = 12 bits

Number of bits required for page directory index: 38 - 14 - 12 = 12 bits

3. Suppose a computer keeps its page tables in main memory. The average overhead required for reading an entry from the page table is 5ns. To reduce this overhead, the computer has a TLB, which holds 32 page table entries, and can do a look up in 1ns. Cache hit rate is the number of hits divided by total accesses. What hit rate is needed to reduce the mean overhead to 2ns?

When the page entry is not in the TLB it takes 1ns for the lookup and 5ns to read from the page table, 6ns total.

When the page entry is in the TLB it takes 1ns. If *hit* is hit rate, then the miss rate is *(1 –hit)*.

2 = 6 * (1 -hit) + 1 * hit
2 = 6 - 6 * hit + hit
4 = 5 * hit
hit = 4/5 = 0.8 hit rate

Modules 6 and 7

1. Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the `bid(amount)` function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
  void bid(double amount) {
    if (amount > highestBid) {
      highestBid = amount;
    }
}
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

There is a race condition on the variable highestBid. Suppose that two people wish to bid on a computer, and the current highest bid is $1,200. Two people then bid concurrently: the first person bids $1,300, and the second person bids $1,400. The first person invokes the bid() function, which determines that the bid exceeds the current highest bid. But before highestBid can be set, the second person invokes bid(), and highestBid is set to $1,400. Control then returns to the first person, which completes the bid() function and sets hightestBid to $1,300. The easiest way of fixing this race condition is to use a mutex lock:

```
void bid(double amount) {
    acquire(mutex);
    if (amount > highestBid)
        highestBid = amount;
    release(mutex);
}
```

2. Consider the code example for allocating and releasing processes.

```
#define MAX_PROCESSES 255
int number_of_processes = 0;
/* the implementation of fork() calls this function */
int allocate_process() {
  int new_pid;
  if (number_of_processes == MAX_PROCESSES) {
    return -1;
  } else {
    /* allocate necessary process resources */
    number_of_processes++;
    return new_pid;
  }
}

/* the implementation of exit() calls this function */
void release_process() {
  /* release process resources */
  number_of_processes--;
}
```

a. Identify the race condition(s).

b. Assume you have a mutex lock named mutex with the operations lock() and unlock(). Indicate where the locking needs to be placed to prevent the race condition(s).

a. There is a race condition on the variable number_of_processes.
b. A call to acquire() must be placed upon entering each function and a call to release() immediately before exiting each function.

3. Assume a network is shared among different services, each of which wants to have multiple concurrently open TCP connections. The network sets a limit on the total number of open TCP connections. The purpose of the methods below is to keep track of the number of open connections so that the limit is never exceeded. For example, if a service want to open 5 new connections it must first call `request(5)`.
Implement the methods (pseudocode is fine) using only **locks** and **condition variables** to manage the concurrency.

```
/* initialize the network capacity to n connections */
void
init(int n);


/* The purpose of this function is to block (not return) until
* n network connections are granted to the caller.
*
* On return, the caller assumes they are allowed to open n
* network connections.
*/
void
request(int n);

/* release n connections */
release(int n);

cond_t cond;
mutex_m mutex;
int available;
void
init(int n) {
  // assuming this is called before the other threads are
  // created, it that is not then case then mutex lock needs to
  // be acquired before updating available
  available = n;
}
void
request(int n) {
  pthread_mutex_lock(&mutex);
  while (n > available) {
    pthread_cond_wait(&cond, &mutex);
  }
  available -= n;
  pthread_mutex_unlock(&mutex);
}

void
release(int n) {
  pthread_mutex_lock(&mutex);
  available += n;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

4. Consider the following sushi bar problem. There is a sushi bar with one chef and N seats. When a customer arrives, they take a seat at the bar if there is an empty one or they leave if no seats are empty. After taking a seat, they wait for the sushi chef to be available and then give their order. After the sushi chef serves them, they eat and leave. Write a solution to the sushi bar problem that uses only **semaphores** to manage the concurrency. You should write three procedures: `init()`, `chef()` and `customer()`.
This problem shares some similarities to a classic problem we didn't cover in class, the sleeping barber problem.
https://en.wikipedia.org/wiki/Sleeping_barber_problem

```
sem_t customersReadyToOrder = 0; /* can be 0 up to N */
sem_t chefTakingOrder; /* will always be 0 or 1 */
sem_t customerGivingOrder; /* will always be 0 or 1 */
sem_t chefServingFood; /* will always be 0 or 1 */
sem_t seatsMutex = 0; /* only one thread at a time allowed
                         to read/write seatsAvailable */
int seatsAvailable = N; /* can be 0 up to N */

void
init() {
    sem_init(customersReadyToOrder, shared, 0);
    sem_init(chefTakingOrder, shared, 0);
    sem_init(customerGivingOrder, shared, 0);
    sem_init(chefServingFood, shared, 0);
    sem_init(seatsMutex, shared, 0);
    int seatsAvailable = N;
}

void chef() {
    while (true) {
        sem_wait(customersReadyToOrder); /* wait for at least one
                                            customer to be ready */
        printf("Welcome to the sushi bar.\n");
        /* take order */
        sem_post(chefTakingOrder);
        printf("What will you have?\n");
        sem_wait(customerGivingOrder);
        /* serve customer */
        printf("Here is your food.\n");
        sem_post(chefServingFood);
    }
}
```

```
void customer() {
    sem_wait(seatsMutex); /* get exclusive access to seatsAvailable
                             to prevent race conditions */
    if (seatsAvailable == 0) {
        sem_post(seatsMutex); /* release the lock */
        printf("I'm not waiting around, bye.\n");
        return;
    }
    printf("This seat looks good.\n");
    seatsAvailable--; /* customer takes an available seat */
    sem_post(seatsMutex); /* release the lock */

    sem_post(customersReadyToOrder); /* customer signals they are
                                        ready to order */
    sem_wait(chefTakingOrder); /* customer waits, the chef might be
                                  serving other customers */

    printf("Hello.\n");

    printf("I will have one roll.\n");
    sem_post(customerGivingOrder);
    sem_wait(chefServingFood);
    printf("That was good.\n");

    set_wait(seatsMutex); /* get exclusive access to seatsAvailable
                             to prevent race conditions */
    seatsAvailable++;
    sem_post(seatsMutex); /* release the lock */
}
```