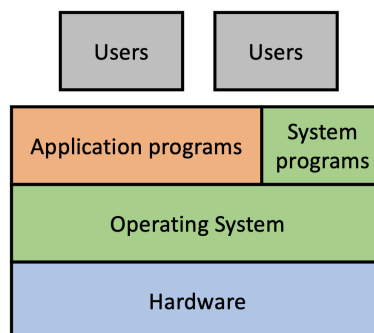# Operating Systems Midterm Review Notes

(Written by Ruixiang Jiang)

Intro to OS

How a Modern Computer Works

- Von Neumann Architecture
    - a single, shared memory for programs and data
    - a single bus for memory access
    - an arithmetic unit
    - a program control unit
- Computer System Components
    - hardware
    - operating system
    - application programs
    - users

What is an OS

- A group of software that makes the computer operate correctly and efficiently in an easy-to-use manner
- It includes a software program called kernel
    - manages all physical devices
    - exposes some functions, such as sys-calls, for others to configure the kernel or build software on top
- It includes other programs such as a shell, a GUI, and a browser
- An OS is a resource manager as well as a control program
- An OS can do the following:
    - virtualization
        - CPU level: run multiple programs on one CPU, as if there are multiple CPUs

- Memory level: give each process or program the illusion of running in its own memory address space
  - concurrency: make sure multi-threaded programs execute correctly
  - persistence: write data into persistent storage

OS Basic Concepts

- Process: a program in execution, executing instructions sequentially, one at a time, until completion
- Process management
  - creating and deleting both user and sys processes
  - suspending and resuming processes
  - process synchronization
  - process communication
  - deadlock handling
- Memory: CPU only directly interacts with main memory (usually DRAM) during execution
- Memory management
  - keeping track of which parts of memory are currently being used and by whom
  - deciding which processes and data to move into and out of memory
  - allocating and deallocating memory space as needed
- Storage management: files abstract physical properties to logical storage units
- IO subsystem
  - memory management of IO
    - buffering
  - general device-driver interface
  - drivers for specific hardware devices

OS Basics

Dual-mode Operations
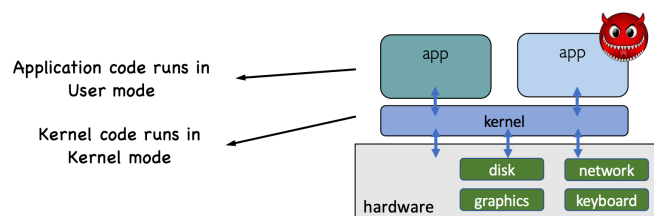
- Alteration in hardware to support dual-mode operations
  - a bit to represent the current mode
  - certain operations only permitted in kernel mode
  - changing from user mode to kernel mode: set kernel mode, save the user PC register

- changing from kernel mode to user mode: transparent kernel mode, restores appropriate user PC register
- Types of mode transitions
  - system call
    - doesn't have the address of the system function to call
    - has a number associated with each system call
    - store the syscall id and args in registers and exec syscall
  - interrupt
    - external
    - asynchronous
    - independent of the user process
  - trap/exception
    - internal
    - synchronous

Kernel Structures

- Monolithic kernel (Linux, Unix, MS-DOS)
  - holds all privileges to access I/O devices, memory, hardware interrupts, and the CPU stack
  - contains many components, such as memory subsystems, I/O subsystems, filesystems, device drivers, etc.
  - higher speed
- Microkernel (Mach, seL4)
  - outsources the traditional operating system functionality to ordinary user processes
  - OS functionalities are pushed to user-level servers
  - better flexibility, security, and fault tolerance
- Hybrid kernel (Windows): combination of a monolithic kernel and a microkernel
- Exokernel: libOS directly manages resources

OS Services

- User interface
  - Command line interface
  - Graphical user interface
- Program execution
- IO operation
- File management
- Communication between processes (via shared memory or through messages passing over a network)

- Error detection and handling

- Resource management

- Memory management

- Process management

- Time management

- Networking

- Protection and security

Processes

Process

- A process is a program in execution

- A process is an abstraction of machine states

- A program is a file on the disk (code and static data)

- A process is loaded by the OS

    - from the program load the code and static data

    - OS creates the heap and stack

- Each process has a unique PID, which can be printed by `getpid()`

System Call

- System call is a function call exposed by the kernel, i.e., abstraction of kernel operations

- System call is different from function call, as it is called by numbers

- System call may need to pass parameters to OS

    - register: pass parameters in registers (limited length or number of parameters)

    - block: pass address of a memory block which contains parameters, as a parameter in a register

    - stack: push parameters onto the stack by the program and pop them by OS

- Library API call is a layer of indirection for system calls

    e.g. `fopen("hello.txt", "w")` is a library call, as `open("hello.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666)` is a system call

Process Creation

- `fork` sys call creates a new process

    - both the parent and the child execute the same program, but the child process starts at the location where that `fork` is returned

    - the return value for the parent and the child is the child's PID and 0, respectively

    - `folk` clones:

        - program counter (CPU register)

- - - program code (file and memory)
    - memory (local/global variables and dynamically allocated memory)
    - opened files (kernel's internal)
  - `folk` doesn't clone:
    - return value
    - PID
    - parent process
    - running time
    - file locks
  - `wait` suspends the calling process to waiting
    - `wait` returns
      - when one of its child processes changes from running to terminated
      - immediately if no child or a child terminates before `wait` is called
    - `wait` the return value is
      - one child is terminated: the PID of the child
      - several children are terminated: the PID of an arbitrary child
      - no child: -1
  - `waitpid` will wait for a particular child only and detect different status changes in the child
- `exec` sys call replaces the process's memory space with a new program
  - `execl`: a member of `exec` syscall family
    - arg1: the file that the programmer wants to execute
    - arg2: arg[0] of the program relative to arg1
    - argn (NULL): the end of the program list
  - in `exec`, the `exit` statement will terminate the process, so the process cannot go back to the old program
  - `exec` replace user-space info:
    - register value (e.g., program counter)
    - program code
    - memory (local/global variables and dynamically allocated memory)
  - `exec` doesn't replace kernel-space info:
    - PID
    - process relationship

## Kernel View of Processes

- Process Control Block (PCB)
  - contains information on each process

- process state
- process number
- program counter
- registers
- memory lists
- opened files
- etc.
 - PCBs are linked in multiple queues
    - ready queue contains all processes in the ready state
    - device queue contains processes waiting for IO events from this device
    - processes may migrate among these queues
- Context Switch
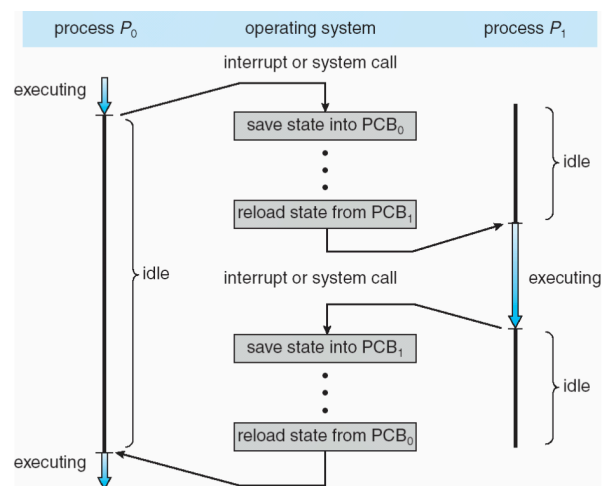
- `fork` in kernel
  - copy PCB for the child
  - add the child PCB to the kernel task list
  - update PCB for both parent and child
  - copy on write: the PCBs of parent and child point to the same region in user space memory
  - return a value
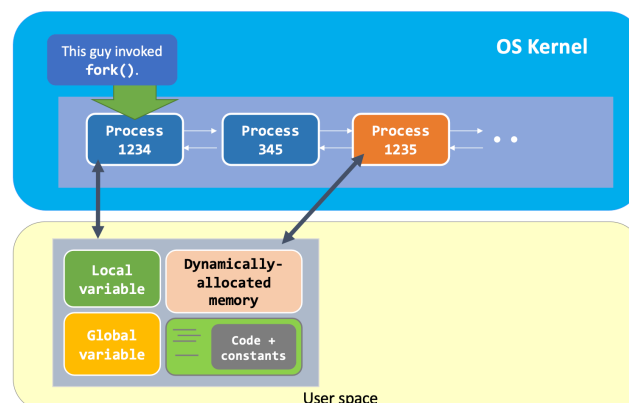
- `exec` in kernel: change sth. in user space

- clear local variables and dynamically allocated memory
  - reset global variables based on the new code
  - reset register values
  - change to the new program code
- `exit` in kernel
  - free most of the allocated kernel space memory (the list of opened files are all closed)
  - free everything on the user space memory
  - when a child calls `exit`, the PID remains in the process table, and the status of the child is zombie; the kernel sends a `SIGCHLD` signal to the parent
  - when a parent calls `exit`, the parent pointer of each child points to `init` process
- `wait` in kernel
  - the parent ignores the `SIGCHLD` by default
  - after the parent calls `wait`, the kernel will register a signal handling routine for the parent
  - when `SIGCHLD` comes (or already there), the corresponding signal handling routine is invoked, and then
    - accept and remove the `SIGCHLD` signal
    - destroy the child process in the kernel space (remove it from process table, etc., which means the child is truly dead)
  - the kernel deregisters the signal handling routine and returns the PID of the dead child as the return value of `wait`
- `init` process
  - while booting up, the kernel creates the `init` process with PID = 1
  - the `init` process will become the re-parent of all orphans
  - the re-parent operation enables background jobs which allows a process runs without a parent terminal/shell

Measure Process Time

- User time: sum of CPU time spent on codes in user space memory, such as calculations, processing data, running algorithms, or invoking a lib
- Sys time: sum of CPU time spent on codes in kernel space memory, such as accessing hardware devices, managing memory, or performing other system-related operations
- Real time: the actual elapsed time as experienced by a human observer, including both the time spent by execution in CPU and external events such as user inputs or IO operations
- Theoretically, the real-time equals user time + sys time
  - real > user + sys: IO intensive
  - real < user + sys: multi-core

Thread

- A process may have multiple threads
- Each thread has its private execution state
    - registers (e.g., program counter)
    - stack
    - context switch
- Threads in the same process share computing resources
    - address space
    - files
    - signals
- Thread implementation
    - user level
        - user level threads library does thread management
        - OS doesn't know about user level thread
    - kernel level
        - kernel does thread management
        - OS knows each kernel level thread
- Thread model
    - many-to-one mapping: many user threads to one kernel thread
        - context switch is cheap between user level threads
        - if a thread blocks, all threads in the process block
    - one-to-one mapping: one user thread to one kernel thread
        - each thread runs or block independently
        - use syscall to cross into kernel mode to schedule
    - many-to-many mapping: many user threads to many kernel thread
        - more user level threads than kernel level threads
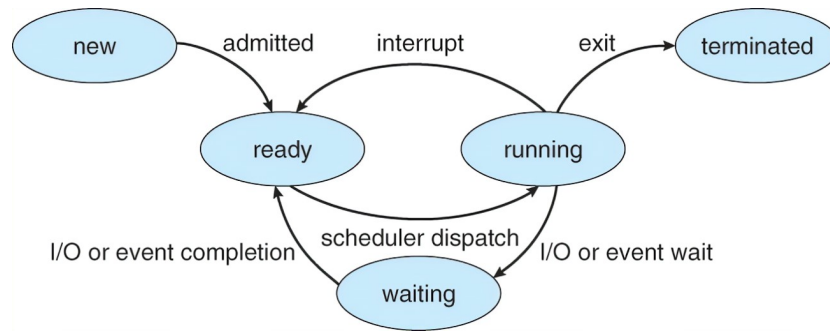   - flexible but challenging to implement

CPU Scheduling

CPU Scheduler

- CPU scheduler selects one of the processes that are ready to execute and allocates the CPU to do it
- The state of a process can be

- CPU scheduler makes the decision when a process
    - switches from running to waiting state
    - switches from running to ready state
    - switches from waiting to ready state
    - terminates

Scheduling Algorithm

- Algorithms take place only when a process switches from running to waiting or terminates is non-preemptive; the other algorithms are preemptive
- Non-preemptive SJF
    - when the CPU becomes available, the scheduler selects the process with the shortest burst time from the ready queue
    - if multiple processes have the same shortest burst time, choose the first arrived one
- Preemptive SJF
    - whenever a process arrives in the ready queue, the scheduler selects the process with the shortest remaining burst time from the ready queue
    - if multiple processes have the same remaining shortest burst time, choose the first arrived one
    - preemptive SJF can achieve the least average waiting time, but it's not very useful due to the difficulty of estimating the remaining burst time
    - practically, use past CPU burst time to estimate the next CPU burst time (e.g., exponential moving average)
- Round Robin
    - each process is given a quantum
    - for a running process, when the quantum is used up, it is preempted and placed at the end of the ready queue
    - new arriving process will be added to the tail of the ready queue without triggering a new selection decision
    - the responsiveness of the processes is great
- Priority Scheduling
    - each process has a priority number
    - for each process, the priority can be

- static: fixed throughout its lifetime

- dynamic: changed over time

- the Priority Scheduling can be

  - non-preemptive: newly arrived process is only put into the ready queue

  - preemptive: A newly arrived process triggers a selection decision

- starvation: low-priority processes may never execute

  aging: increase the priority as time goes by

- Completely Fair Scheduler (Linux)

  - CFS uses a red-black tree data structure to make sure that each process has the same run time

  - each node is a process with a virtual run time as the key value

  - each process has a nice value (-20 to 19) as the priority; the lower the nice value, the higher the priority, and the higher the proportion of CPU time

    e.g., a process has a nice value of 0, and its weighted value is 1024, another process whose nice value is -1 has a weighted value of 1277, then the first process has a proportion of CPU time $\frac{1024}{1024+1277} = 44.5\%$, and the running time is $\frac{44.5\%}{50\%}$ of its vruntime theoretically
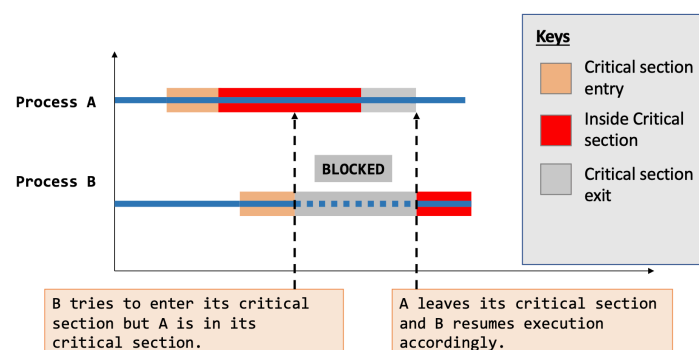
Synchronization

Race Condition

- Threads may share global variables, while processes may share the same region of memory

- For non-atomic instructions, the execution result depends on the execution sequences

Solution 1: Mutual Exclusion

When a process accesses the shared memory, the other processes can not access it

Solution 2: Disabling Interrupts

- Disable interrupts when the process is in the critical section

- For uni-core: correct but not permissible

- For multi-core: incorrect unless turn off interrupts on all cores

Solution 3: Locks

- Basic idea: use another shared object and atomic instructions

- Spin-based lock

  - process level

    - basic: use a shared object `turn` to detect the status of other processes
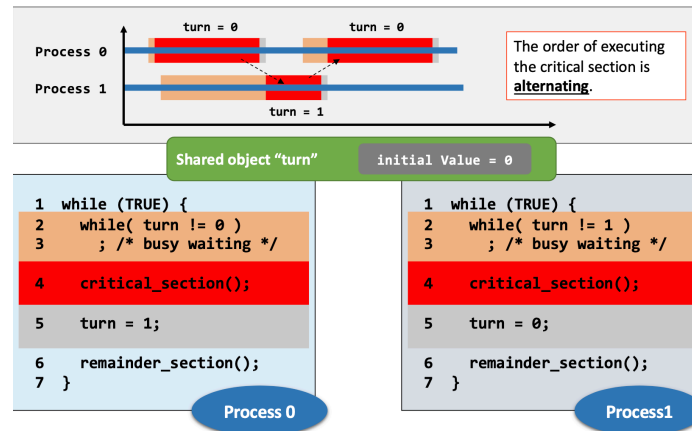
    - correct but wastes CPU resources

      spin lock needs spin-time, but sleep lock needs context switch, so the spin lock is ok only for spin-time < context-switch-overhead

    - cause progress violation if a process has a very slow remainder_section

      - the process gives out the `turn` and now is in the remainder part

      - then another process finishes the critical section, gives back the `turn` to the first process, finishes its remainder part, and enters the busy waiting part

      - finally, the first process is in the remainder part, as another process is waiting

  - Peterson's solution

```
1   // Process i
2   interested[i] = True;
3   turn = j;
4   while (interested[j] && turn == j);
5   critical_section();
6   interested[i] = False;
7   remainder_section_i();
8
9   // Process j
10  interested[j] = True;
11  turn = i;
12  while (interested[i] && turn == i);
13  critical_section();
14  interested[j] = False;
15  remainder_section_j();
16
17  /* Example
18  originally, turn = i and interested[] = {False}
19  as for process i:
20      it wants to enter the critical section (interested[i] = True)
21      but it declines out of modesty (turn = j)
```

```
22        it enters the wait while loop
23   as for process j:
24        it is also interested in the critical section but modest, setting
     turn = i
25   as for process i:
26        it finds that turn = i, which contradicts the while condition, so
     goes out of the while loop and enters the critical section
27   */
```

- as for the progress violation, a process in the remainder part is not interested in executing the critical section, but it has the `turn`
- use an `interested` array to avoid it
- to prove its correctness, show that
    - mutual exclusion is preserved, i.e., at most one process can enter the critical section at any time

      Prove by contradiction: suppose two processes, i and j, enter the critical section at the same time; at least we have `interested[i]` = `interested[j]` = True.

      Note that `turn` can only be i or j, but not both. Suppose process i enters the critical section, which means it breaks the waiting while loop; it follows that `turn` = i.

      Hence, process j must be in a waiting-while loop as long as process i is in its critical section, i.e., mutual exclusion is preserved.
    - the progress requirement is satisfied, i.e., at most one process can be selected to enter the critical section at any time

      If only one process is interested in the critical section, its dream will come true.

      If all the processes are interested in the critical section, note that the `turn` has only one value, which means only one process will be selected.
    - the bounded-waiting requirement is met, i.e., a process will enter the critical section after at most one entry by another process

      Suppose both process i and j want to enter the critical section, and process i is selected first. After process i exits the critical section, process j is waiting. Before process j enters the critical section, there are two cases.

      If process i executes its remainder part very fast, it will set `interested[i]` = True and `turn` = j, which breaks the waiting while loop for process j.

      If process i executes its remainder part not so fast, it will set `interested[i]` = False, which also breaks the waiting while loop for process j.

      Hence, process j must enter the critical section, i.e., process j will enter the critical section after at most one entry by process i.

      In other words, after process i exits the critical section, it can not enter it again before process j realizes it.
- multi-process mutual exclusion

```
1  // Process i
2  waiting[i] = True;
3  key = True;
4  while (waiting[i] && key) key = test_and_set(&lock);
5  waiting[i] = False;
6  critical_section();
7  j = (i + 1) % n;
8  while (j != i && !waiting[j]) j = (j + 1) % n;
9  if (j == i) lock = False; else waiting[j] = False;
```
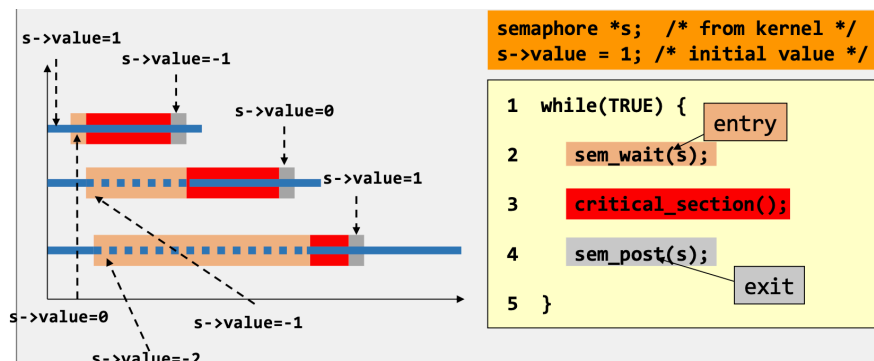
- thread level
- Sleep-based lock
  - Semaphore

    semaphore is a struct with an integer `value` and a queue-based list `process_id`.

```
1  // Initially
2  semaphore *s;
3  s->value = 1;
4
5  //Process i
6  if (--s->value < 0) sleep(); // also named wait
7  critical_section();
8  if (++s->value <= 0) wakeup(); // also named post
```


must guarantee that two processes can not execute `s->value -= 1` and `s->value += 1` at the same time; that is another critical section problem

- for single-processor machine: turn off the interrupt
- for multi-core architecture: use atomic instruction `cmp_xchg`

```
1  void cmp_xchg(int *addr, int expected_value, int new_value){
2      int temp = *addr;
3      if (*addr == expected_value) *addr = new_value;
4      return temp;
5  }
6
7  // use cmp_xchg to implement increment for s->value += 1
8  do{
9      int old = *addr;
```

```
10        int new = old + 1;
11   } while (cmp_xchg(addr, old, new) != old);
12
13   /*
14   if *addr = old, it means no critical section problem, so that update
     *addr to *addr + 1 and quit the do-while loop
15   if *addr != old, it means *addr has been changed unexpectedly, so that
     do nothing in this loop, and do the next loop
16   */
```

- Producer-Consumer Problem Solved by Semaphore

  There is a pipe with a fixed length. The producer pushes data to the tail of the pipe, while the consumer pops data from the head of the pipe.

  - requirement 1: when the producer wants to push a new item but the pipe is full, it should wait; the consumer should notify the producer after popping an item.

    i.e., notify the producer to push when the pipe is not full

  - requirement 2: when the consumer wants to pop a new item but the pipe is empty, it should wait; the producer should notify the consumer after pushing an item.

    i.e., notify the consumer to pop when the pipe is not empty

```
1    // shared object area
2    // abstraction of semaphore as an integer
3    semaphore mutex = 1; // used as a buffer lock to avoid race condition
4    semaphore avail = pipe_size;
5    semaphore filled = 0; // avail and filled are used to achieve
     synchronization
6
7    // Producer
8    while (1){
9        item = produce_item();
10       wait(&avail); // wait for an empty slot and acquire to push an item
11       wait(&mutex); // prevent the consumer to access the buffer
12       insert_item(item);
13       post(&mutex); // the consumer can access the buffer now
14       post(&filled); // notify the consumer that the buffer is filled
15   }
16
17   // Consumer
18   while (1){
19       wait(&filled); // wait for an item and acquire to pop it
20       wait(&mutex); // prevent the producer to access the buffer
21       item = remove_item();
22       post(&mutex); // the producer can access the buffer now
23       post(&avail); // notify the producer that the buffer has an empty slot
     now
24       consume_item(item);
25   }
```

- if remove the semaphore `filled`, it raises an error when the consumer gets CPU first because the consumer will try to pop an item from an empty buffer
- if exchange `wait(&avail)` and `wait(&mutex)` before inserting an item in the producer part, it causes endless sleep when the producer gets CPU to keep producing until the buffer is full because after filling up the buffer, the producer will sleep due to `wait(&avail)`, as it has executed `wait(&mutex)` before. now the `mutex` value is -1 less than 0, which means the consumer should be waiting in `wait(&mutex)`, but no one can wake it up, i.e., an endless sleep.

- Dining Philosopher Problem

  5 Philosophers are engaged in two activities: Thinking and Eating. Meals are taken communally on a table with five plates and five forks cyclic. Every Philosopher needs two forks to eat.

  - requirement 1: no two Philosophers can have the same two forks simultaneously (mutual exclusion)
  - requirement 2: each philosopher can get the chance to eat in a particular finite time (deadlock)
  - requirement 3: when few Philosophers are waiting, then one gets a chance to eat in a while (starvation)
  - solution 1 with deadlock

    ```
     1  // for Philosopher i
     2  // initially semaphore array fork[] is {1}
     3  while (1){
     4      think();
     5      wait(fork[i]);
     6      wait(fork[i + 1]);
     7      eat();
     8      post(fork[i]);
     9      post(fork[i + 1]);
    10  }
    11  // this solution may lead to a deadlock under an interleaving that has
        all the Philosophers pick up their left forks before any of them tries
        to pick up a right fork. In this case, all the Philosophers are waiting
        for the right fork but no one will execute a single instruction.
    ```

  - solution 2 with starvation

    firstly, each Philosopher takes a fork. if the Philosopher finds that he can not take another fork, he should put down the fork in his hand and then sleep for a while. when they wake up, the Philosopher tries again to take both forks.

  it causes starvation if all Philosophers take a fork first and then put down the forks and sleep simultaneously. they wake up and do this loop again with starvation.

  - final solution

    ```
    1  #define N 5
    2  #define LEFT ((i + N - 1) % N)
    ```

```
3   #define RIGHT ((i + 1) % N)
4
5   int state[N]; // state of each philosopher, including EATING, THINKING
    and HUNGRY
6   semaphore mutex = 1; // guarantees mutual exclusive access to state
    array
7   semaphore p[N] = 0; // models a philopher
8
9   void captain(int i){
10      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=
    EATING){
11          state[i] = EATING;
12          post(&p[i]);
13      }
14  }
15
16  void philosopher(int i){
17      think();
18
19      wait(&mutex);
20      state[i] = HUNGRY;
21      captain(i);
22      post(&mutex);
23      wait(&p[i]);
24
25      eat();
26
27      wait(&mutex);
28      state[i] = THINKING;
29      captain(LEFT);
30      captain(RIGHT);
31      post(&mutex);
32  }
```

- Read Writer Problem

  A set of data is shared among several processes. Once a writer is ready, it performs its write. Only one writer may write at a time. If a process is writing, no other process can read it. If at least one reader is reading, no other process can write. There are no limitations on the number of readers reading simultaneously. There are three situations according to the priorities of the readers and the writer.

  - reader has the higher priority

    Suppose some readers are reading and a writer is waiting; if a new reader comes, it can directly enter reading ahead of the waiting writer.

```
1   semaphore wrt = 1; // ensure mutual exclusion for critical section
2   semaphore mutex = 1; // ensure mutual exclusion when readcnt is updated
3   int readcnt = 0; // the number of readers now
4
5   void write(){
```

```
6        do{
7            wait(wrt); // writer requests for critical section
8            write();
9            post(wrt); // writer leaves the critical section
10       } while (true);
11   }
12
13   void reader(){
14       do{
15           wait(mutex); // protect readcnt
16           readcnt++; // a new reader comes
17           if (readcnt == 1) wait(wrt); // only the first reader should
     prevent the writer from entering the critical section
18           post(mutex); // unlock readcnt so that other readers can enter
19
20           read();
21
22           wait(mutex); // the reader wants to leave
23           readcnt--;
24           if (readcnt == 0) post(wrt); // there is no readers now so the
     writer is able to enter
25           post(mutex);
26       } while (true);
27   }
```

- writer has the higher priority

  Suppose some readers are reading and a writer is waiting; no readers can enter reading, and the writer will enter the critical section as long as all the readers leave.

```
1    int readcnt = 0, writecnt = 0;
2    semaphore rmutex = 1, wmutex = 1; // protect the readcnt and writecnt
3    semaphore readTry = 1; // whether readers can enter, i.e., whether there
     are writers waiting to enter
4    semaphore resource = 1; // ensure mutual exclusion for critical section
5
6    void reader(){
7        wait(readTry); // a reader is trying to enter
8        wait(rmutex); // protect readcnt
9        readcnt++;
10       if (readcnt == 1) wait(resource); // the first reader should lock
     the resource
11       post(rmutex);
12       post(readTry);
13
14       read();
15
16       wait(rmutex);
17       readcnt--;
18       if (readcnt == 0) post(resource); // the last reader should release
     the locked resource
19       post(rmutex);
```

```
20   }
21
22   void writer(){
23       wait(wmutex); // protect writecnt
24       writecnt++;
25       if (writecnt == 1) wait(readTry); // the first writer should lock
     the resource
26       post(wmutex);
27       wait(resource); // prevent other writers from editing the shared
     resource simultaneously
28
29       write();
30
31       post(resource);
32       wait(wmutex);
33       writecnt--;
34       if (writecnt == 0) post(readTry); // the last writer should unlock
     the resource
35       post(wmutex);
36   }
```

- reader and writer have the same priority

  It preserves the ordering of requests. Use an administrator to control.

```
1    int readcnt = 0;
2    semaphore resource = 1; // control access to the resource, which is
     actually binary
3    semaphore rmutex = 1; // protect readcnt
4    semaphore serviceQueue; // preserve the ordering of requests
5
6    void reader(){
7        wait(serviceQueue);
8        wait(rmutex);
9        readcnt++;
10       if (readcnt == 1) wait(resource); // the first reader should request
     resource access for readers (writers blocked)
11       post(serviceQueue);
12       post(rmutex);
13
14       read();
15
16       wait(rmutex);
17       readcnt--;
18       if (readcnt == 0) post(resource);
19       post(rmutex);
20   }
21
22   void writer(){
23       wait(serviceQueue);
24       wait(resource);
```

```
25        post(serviceQueue); // the ordering here from wikipedia is a bit
      strange
26
27        write();
28
29        post(resource);
30    }
```

WELCOME REWARD!