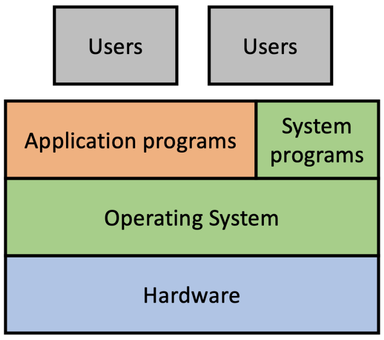
Operating Systems Review Notes

(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 operates 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 following:
    - virtualization
        - CPU level: run multiple programs on one CPU, as if there are multiple CPUs

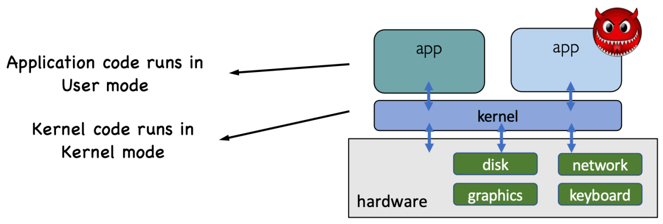
        - memory level: give each process or program the illusion of running in its own memory address space
    - concurrency: make sure multi-threaded programs execute correctly
    - persistence: write data into persistent storage

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


Application code runs in User mode
Kernel code runs in Kernel mode
app
app
kernel
hardware
disk
network
graphics
keyboard

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

    - changing from kernel mode to user mode: clear 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 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
- Micro kernel (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 micro kernel
- Exokernel: libOS directly manages resources

OS Services

- User interface
    - Command line interface
    - Graphical user interface
- Program execution
- IO operation
- File management
- Communication between process (via shared memory or through message 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 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

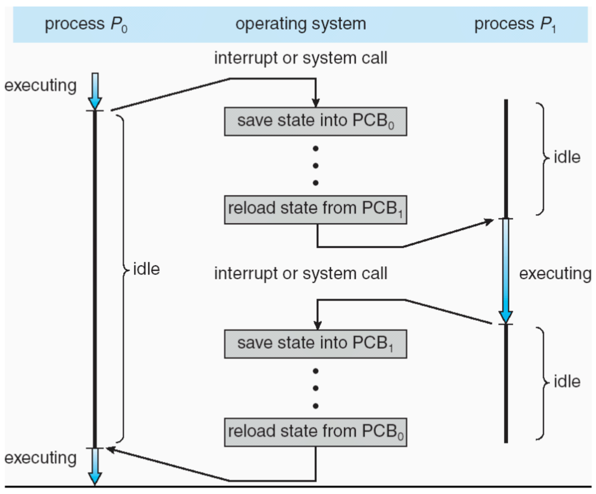
- `fork` sys call creates new process
  - both the parent and the child execute the same program, but the child process starts at the location that `fork` is returned
  - the return value for the parent and the child is child's PID and 0, respectively
  - `folk` clones:
    - program counter (CPU register)

        - 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` 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 of 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

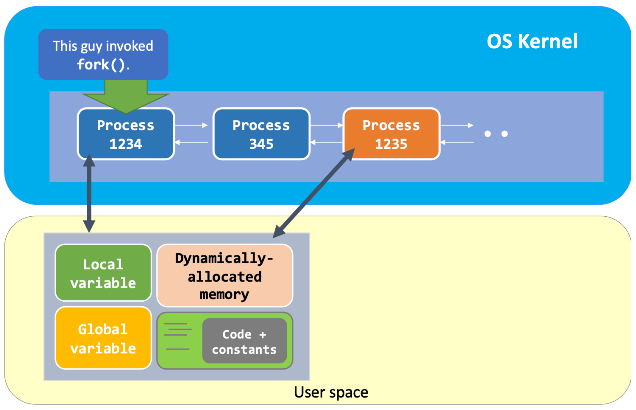
Kernel View of Processes

- Process Control Block (PCB)
    - contains information of 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

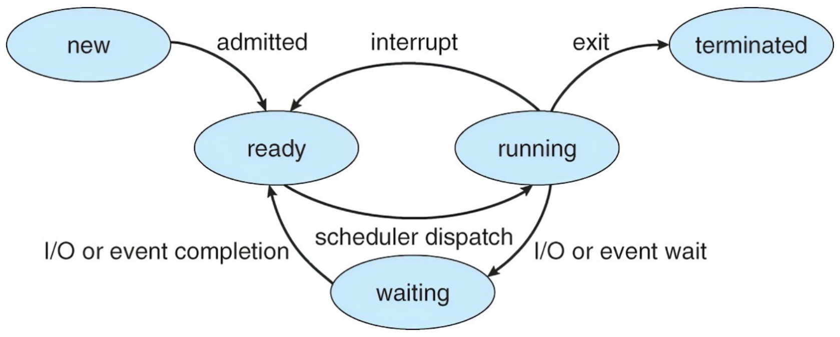
Thread

- A process may have multiple threads
- Each thread has its own 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 know each kernel level thread
- Thread model
    - many-to-one mapping: many user thread 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 run or block independently
        - use syscall to cross into kernel mode to schedule
    - many-to-many mapping: many user thread to many kernel thread
        - more user level threads than kernel level threads
        - flexible but difficult 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 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 firstly 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 firstly arrived one
- 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 triggerring 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: newly arrived process triggers a selection desicion
    - 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 nice value (-20 to 19) as the priority, the lower nice value, the higher priority, the higher proportion of CPU time

      e.g. a process has a nice value 0, and its weighted value is 1024, another process whose nice value is -1 has a weighted value 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

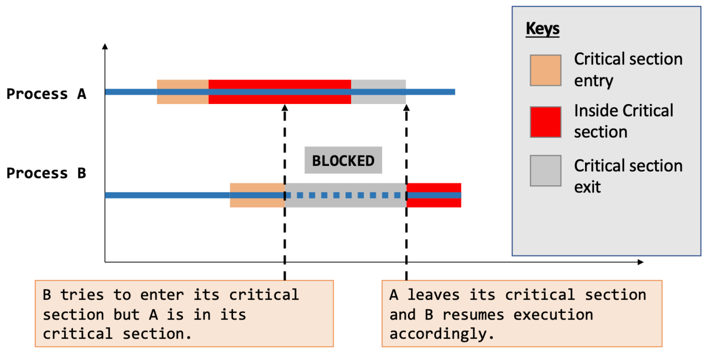
Synchronization

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

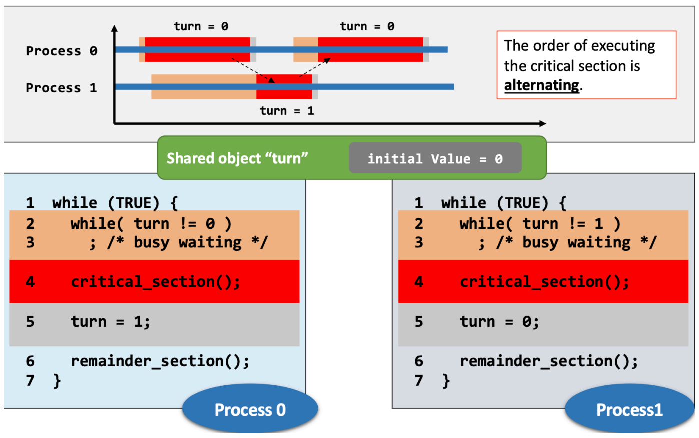

Process A
Process B
BLOCKED
Keys
Critical section entry
Inside Critical section
Critical section exit
B tries to enter its critical section but A is in its critical section.
A leaves its critical section and B resumes execution accordingly.

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 disable 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 process



```
while (TRUE) {
   while( turn != 0 )
     ; /* busy waiting */
   critical_section();
   turn = 1;
   remainder_section();
}
```

```
while (TRUE) {
   while( turn != 1 )
     ; /* busy waiting */
   critical_section();
   turn = 0;
   remainder_section();
}
```

    - correct but waste CPU resources
    - cause progress violation if a process has a very slow remainder_section
        - the process gives out the `turn` and now is in remainder part
        - then another process finishes critical section, gives back the `turn` to the first process, finishes its remainer part, and enters busy waiting part
        - finally the first process is in remainder part, as another process is waiting
- Peterson's solution

```
// Process i
interested[i] = True;
turn = j;
while (interested[j] && turn == j);
critical_section();
interested[i] = False;
remainder_section_i();

// Process j
interested[j] = True;
turn = i;
while (interested[i] && turn == i);
critical_section();
interested[j] = False;
remainder_section_j();

/* Example
originally, turn = i and interested[] = {False}
as for process i:
    it wants to enter the critical section (interested[i] = True)
    but it declines out of modesty (turn = j)
    it enters the wait while loop
as for process j:
    it is also interested in the critical section but modest, setting turn = i
as for process i:
    it finds that turn = i, which contradicts the while condition, so goes out of the while loop and enters the critical section
*/
```

        - as for the progress violation, a process in 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 process 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 either 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 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 brokes 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 brokes the waiting while loop for process j.

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

              In another word, after process i exit the critical section, it can not enter it again before process j realizes it.
    - multi-process mutual exclusion

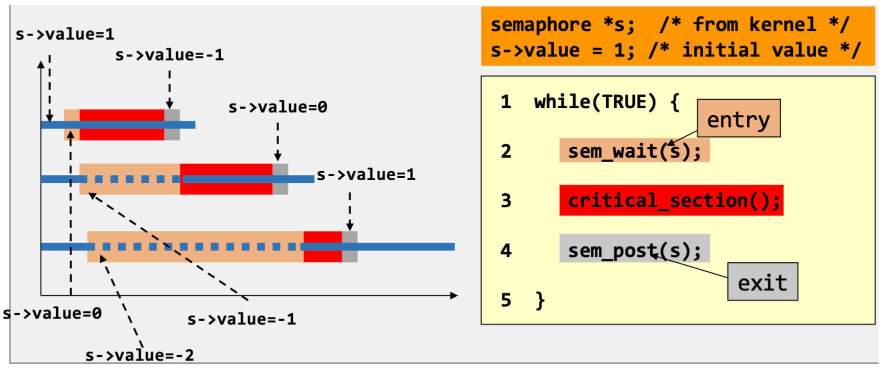
```
// Process i
waiting[i] = True;
key = True;
while (waiting[i] && key) key = test_and_set(&lock);
waiting[i] = False;
critical_section();
j = (i + 1) % n;
while (j != i && !waiting[j]) j = (j + 1) % n;
if (j == i) lock = False; else waiting[j] = False;
```

    - thread level
- Sleep-based lock
    - Semaphore

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

```c
// Initially
semaphore *s;
s->value = 1;

//Process i
if (--s->value < 0) sleep(); // sem_wait
critical_section();
if (++s->value <= 0) wakeup(); // sem_post
```



```c
semaphore *s;  /* from kernel */
s->value = 1; /* initial value */
```

```c
while(TRUE) {
  sem_wait(s);
  critical_section();
  sem_post(s);
}
```

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: disable interrupt
  - for multi-core architecture: use atomic instruction `cmp_xchg`

```c
void cmp_xchg(int *addr, int expected_value, int new_value){
    int temp = *addr;
    if (*addr == expected_value) *addr = new_value;
    return temp;
}

// use cmp_xchg to implement increment for s->value += 1
do{
    int old = *addr;
    int new = old + 1;
} while (cmp_xchg(addr, old, new) != old);

/*
if *addr = old, it means no critical section problem, so that update *addr to *addr + 1 and quit the do-while loop
if *addr != old, it means *addr has been changed unexpectedly, so that do nothing in this loop, and do the next loop
*/
```

- Producer-Consumer Problem Solved by Semaphore

  There is a pipe with 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 poping an item.

    i.e., notify the poducer 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

```
// shared object area
// abstraction of semaphore as an integer
semaphore mutex = 1; // used as a buffer lock to avoid race condition
semaphore avail = pipe_size;
semaphore filled = 0; // avail and filled are used to achieve
synchronization

// Producer
while (1){
    item = produce_item();
    wait(&avail); // wait for an empty slot and acquire to push an item
    wait(&mutex); // prevent the consumer to access the buffer
    insert_item(item);
    post(&mutex); // the consumer can access the buffer now
    post(&filled); // notify the consumer that the buffer is filled
}

// Consumer
while (1){
    wait(&filled); // wait for an item and acquire to pop it
    wait(&mutex); // prevent the producer to access the buffer
    item = remove_item();
    post(&mutex); // the producer can access the buffer now
    post(&avail); // notify the producer that the buffer has an empty slot
now
    consume_item(item);
}
```

  - 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 up it, i.e., an endless sleep.

- Dining Philosopher Problem

  There are 5 Philosophers who are engaged in two activities Thinking and Eating. Meals are taken communally in a table with five plates and five forks in a cyclic manner. Every Philosopher needs two forks in order 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 certain 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

    ```
    // for Philosopher i
    // initially semaphore array fork[] is {1}
    while (1){
        think();
        wait(fork[i]);
        wait(fork[i + 1]);
        eat();
        post(fork[i]);
        post(fork[i + 1]);
    }
    // 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 wake up, the Philosopher tries again to take the both forks.

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

  - final solution

- Read Writer Problem