Philosophers

Mandatory part

To know:

1. **No** global variables :)
2. Arguments:
   - **number of philosophers** = number of forks
   - **time to die :** the philosopher must eat before time_to_die ms after starting or his last meal
   - **time to eat:** in ms, the time takes for a philo to eat.
   - **time to sleep:** in ms it s resting time.
   - **number of times each philo must eat:** optional, if all philosophers eat at least number_of_... times the program stops if it’s not specified the program ends when a philo dies RIP.

Tasks:

Now after we got the arguments we need to start by few things:

1. Each philosopher must have an index between [1 - number_of_philo]

💡 Philo’s index starts from 1 not 0

2. Any change must be written this way:
   - timestamp_in_ms X has taken a fork
   - timestamp_in_ms X is eating
   - timestamp_in_ms X is sleeping
   - timestamp_in_ms X is thinking
   - timestamp_in_ms X died

   X: the philosopher’s index.

   💡 The status printed should not be intertwined with another philo’s status. in other words things must be synchronized. **(It’s all about synchronization)**

3. We can’t have more than **10 ms** between a philo’s death and when it will print its death.
4. We should avoid a philo’s death.

How ?!

1. Each philosopher must be a thread *(you’ll know how to create a thread later).*
2. One fork between each philosopher, if there are multiple philosophers then one fork on the right and another one on the left.
3. To avoid philosophers duplicating forks we need to protect each fork with a mutex.

   How again?!

   → We’ll define at the beginning [number_of_philo mutexes], each philosopher has its own.

   → Once a philosopher picks up a fork, it’s locked using `mutex_lock()`, no one else can use it until it is unlocked by the same philosopher using `mutex_unlock()` and so on.

   → When the program is about to end (a philosopher dies or they all ate at least [number of times to eat] times we have to destroy all the mutexes using `mutex_destroy()`, it’s basically like freeing that object so it can be reinitialized.

Threads:

what is a thread?

A thread in computer science is short for a thread of execution. Threads are a way for a program to divide (termed "split") itself into two or more simultaneously (or pseudo-simultaneously) running tasks.

Threads and processes help us in multitasking but there is a difference between them:

→ using multiple processes gives us two benefits.

Concurrency

Isolation: the memory is separated and if one crushes, the others just keep running.

→ threads offer us only concurrency plus every thread has its own call stack.

Creation:

```
#include <phtreads.h>
```

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg);
```

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr**: pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.

THREAD ATTRIBUTE STRUCTURE:

```
typedef struct {
    int                 __flags;
    size_t              __stacksize;
    void                *__stackaddr;
    void                (*__exitfunc)(void *status);
    int                 __policy;
    struct sched_param  __param;
    unsigned            __guardsize;
                } pthread_attr_t;
```

- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

💡 The last argument can help us to pass arguments to the function but we can also get results from the function using the second parameter of `pthread_join`

Waiting:

```
int pthread_join(pthread_t th, void **thread_return);
```

Used to wait for the termination of a thread.

th: The target thread whose termination you're waiting for.

thread_return*:* NULL, or a pointer to a location where the function can store the value passed to *pthread_exit()* by the target thread.

`pthread_join` does two things:

1. Wait for the thread to finish.
2. Clean up any resources associated with the thread

If you exit the process without joining, then (2) will be done for you by the OS (although it won't do thread cancellation cleanup, just nuke the thread from orbit), and (1) will not. So whether you need to call `pthread_join` depends on whether you need (1) to happen.

If you don't need the thread to run, then as everyone else is saying you may as well detach it. A detached thread cannot be joined (so you can't wait on its completion), but its resources are freed automatically if it does complete.

Detaching:

```
int pthread_detach(pthread_t thread);
```

Used to detach a thread. A detached thread does not require a thread to join on terminating (it s no longer joinable). The resources of the thread are automatically released after terminating if the thread is detached.
In other words, if a thread is detached, it does not need to join back into its parent to release its resources *(and neither can you join it)*. This is useful when you want to have a thread that just does its thing without joining back with its creator.

💡 The thread by default is created joinable but we can make it detached since creation using the thread attributes.

As we said the detached thread can no longer be joinable so the program might exit before its termination to solve this use: `pthread_exit();`

Mutexes *(mutual exclusion)*

The mutex lock allows the thread to lock a code section *(it says: Hey! I have the floor)* and exclude all the other threads.

Initialization:

```
# include <pthread.h>

pthread_mutex_t *mutex;
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

You have to have allocated memory for it ahead of time *(pthread_mutex_init() does not call malloc().)*

Lock:

```
int pthread_mutex_lock (pthread_mutex_t * mutex);
```

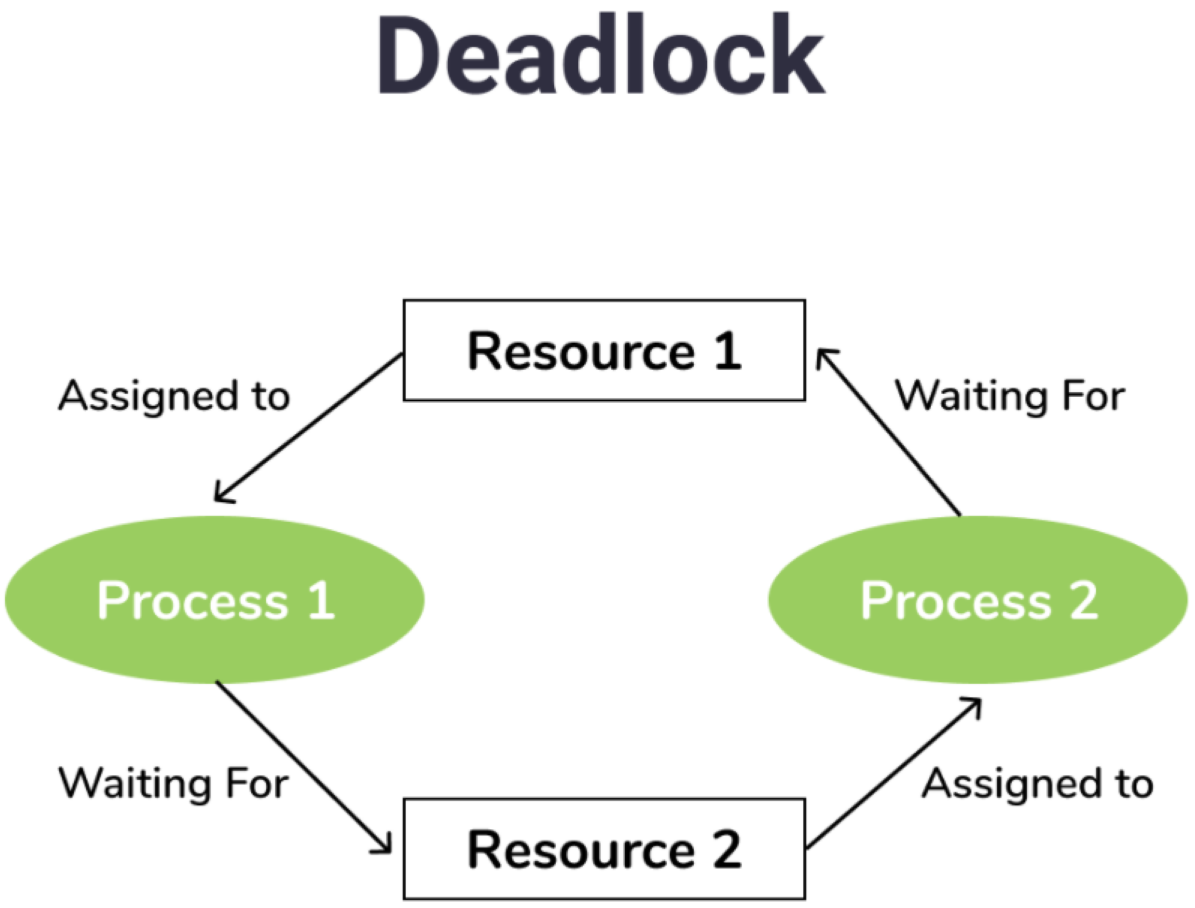
> I want to point out here, that `pthread_mutex_lock()` does not actively **"lock"** other threads. Instead, *it locks a data structure*, which can be shared among the threads. The locking and unlocking of the data structure make synchronization guarantees, which are very important to avoiding *race conditions(see explanation below)*. However, I don't want you to get into the habit of thinking that `pthread_mutex_lock()` actively blocks other threads, or "locks them out." It doesn't -- it locks a data structure, and when other threads try to lock the same data structure, they block. Please re-read this paragraph.

What is a Race Condition?

A race condition occurs when two or more threads can access shared data and try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

What is a Deadlock?

A deadlock happens when two threads/processes cannot make progress because each one waits for the other to release a lock, as illustrated in the following diagram.



Note: if the mutex is locked by a thread, only that thread can unlock it, other threads wait until then.

Unlock:

```
int pthread_mutex_unlock (pthread_mutex_t * mutex);
```

Destruction:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

This function sets the mutex to an invalid value so it can be reinitialized but the destroyed mutex can no longer be used.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy locked mutex results in undefined behavior.

💡 Every mutex must eventually be destroyed with pthread_mutex_destroy(). The machine eventually detects the error if a mutex is not destroyed, but the storage is deallocated or corrupted.

BONUS PART

TODO :

- [ ] Read about fork and multiprocessing
    - [ ] Check the use of the following functions:
        - `fork()`
        - `kill()`
        - `waitpid()`
    - [ ] Create a process for each philosopher.

- [ ] Read about Semaphores
    - [ ] Check the use of the following functions:
        - `sem_open()`
        - `sem_close()`
        - `sem_post()`
        - `sem_wait()`
        - `sem_unlink()`
    - [ ] Create a semaphore that represents the number of available forks, decrement when a Philo takes a fork, and increment when a Philo puts down a fork.

- [ ] Combine the two and try to create a 3 Philos and 3 forks with only the following actions:
    - [ ] Grab a fork with the right hand

    - [ ] Grab a fork with the left hand
    - [ ] Start eating
    - [ ] Finish eating

- [ ] Handle the optional argument [number_of_times_each_philosopher_must_eat]

- [ ] If the above worked perfectly, add the following actions:
    - [ ] Thinking
    - [ ] Sleeping

- [ ] Handle the philosopher's death.

`int fork();`

This system call is used to create one or many child processes.

return:

If it's the main process (the parent) it returns the child pid

If it's the child process it returns 0

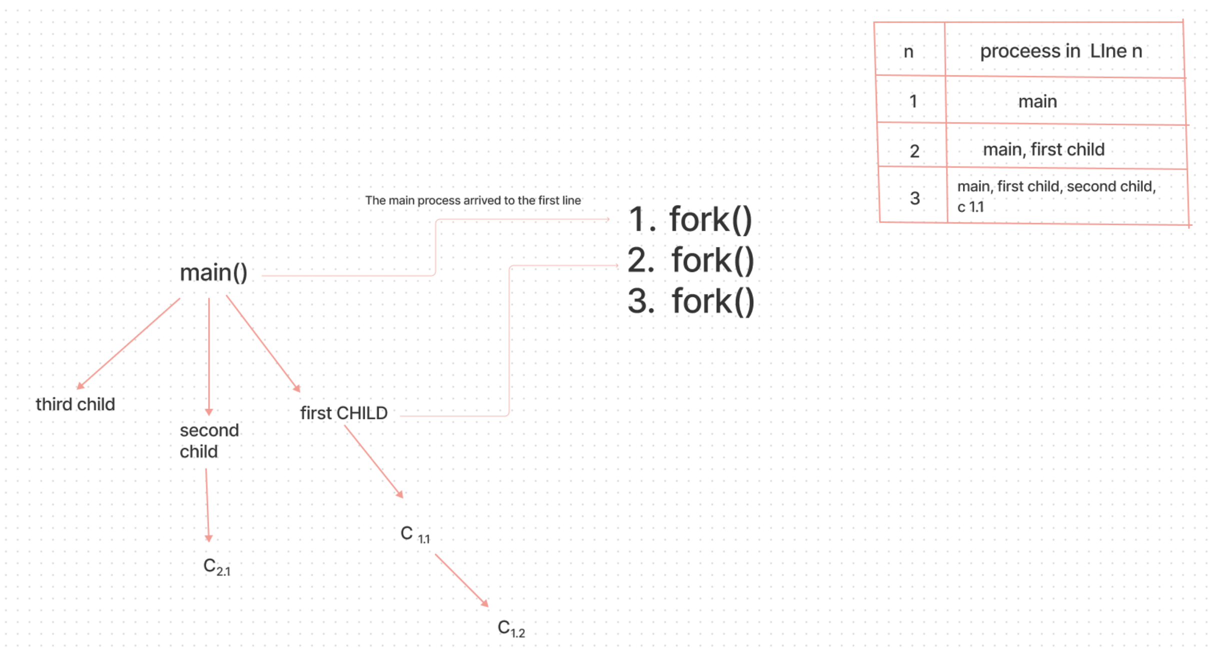
> You might ask why the return value is like we said above, simply because the child won't need the id to be returned it can get simply get it using `getpid()`, but the parent needs to keep track of the children so it needs the ids.

▼ How does it work?

When we call fork the execution line splits into 2 execution lines, the child process created continue the execution from the line forward

$$ProcessesNumber = 2^n (n = CallsNumber)$$

→ Example with 3 calls of fork(), we got 8 processes.



n	proceess in LIne n
1	main
2	main, first child
3	main, first child, second child, c 1.1

▼ How to create a fixed number of processes?

```
int main()
{
  int ret;
  int i;

  i = 0;
  while (i < 5)// we want to create 5 children.
  {
    ret = fork();
    if(ret == 0)
    {
    /*
     ... code to be executed by the child process
    */
    exit(0); // the child peocess exits after completing the task demanded.
    }
    i++;
  }
  return (0);
}
```

`int kill(pid_t pid, int sig);`

```
#include <sys/types.h>
#include <signal.h>
```

This function is used to send signals to a process or a group of signals.

`pid_t waitpid(pid_t pid, int * status, int options);`

```
#include <sys/types.h>
#include <sys/wait.h>
```

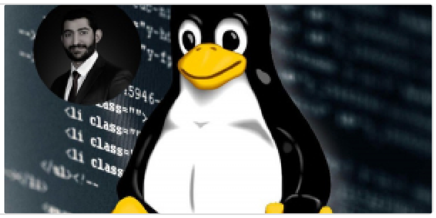
The `waitpid()` system call suspends execution of the current process until a child specified by *pid* argument has changed state. By default, `waitpid()` waits only for terminated children, but this behavior is modifiable via the *options*
 the argument, as described below.

Link for more informations :

waitpid() - Unix, Linux System Call

waitpid() - Unix, Linux System Calls Manual Pages (Manpages) , Learning fundamentals of UNIX in simple and easy steps : A beginner's tutorial containing complete knowledge of Unix Korn and Bourne Shell and Programming, Utilities, File

https://www.tutorialspoint.com/unix_system_calls/waitpid.htm



▼ `int *status`

This argument helps us to know the status of the terminated process

We can use it to check if one of the processes (a philosopher) died, if the status ≠ 0, we can terminate all the other processes and terminate the program.

💡 We can wait for any child instead of a special one: `waitpid(-1, &ret, 0)`;

SEMAPHORE

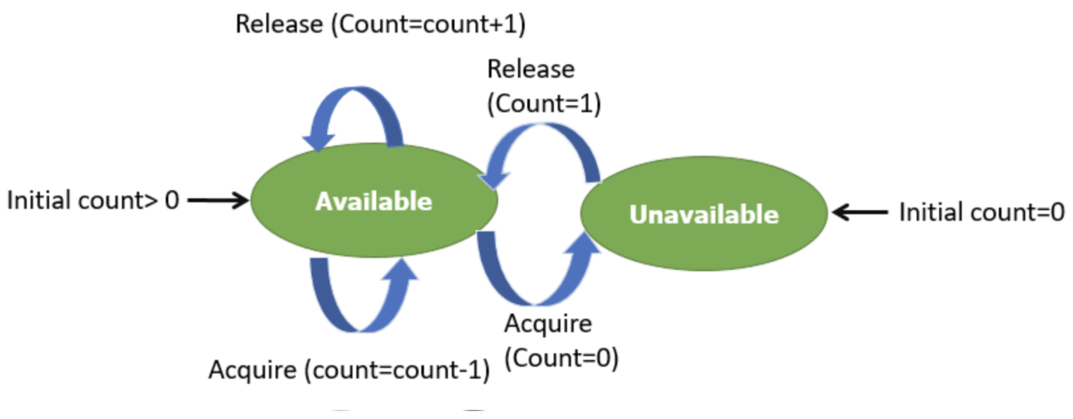
▼ What is a semaphore?

Semaphore is simply an unsigned int. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are of two types:

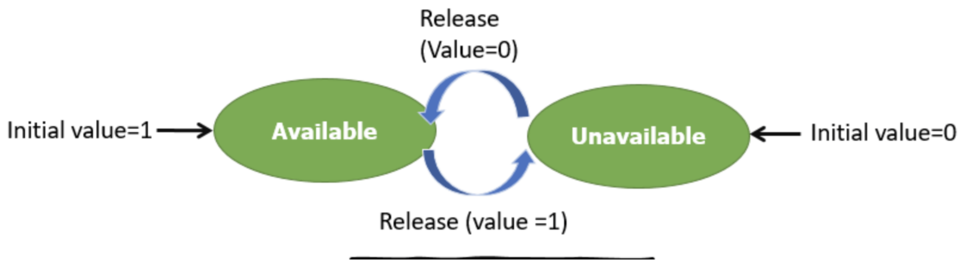
1. **Counting Semaphore**

   – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.



2. **Binary Semaphore**

   – This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.



Note: *there are named and unnamed semaphores.*

Named Semaphores are like process-shared semaphores, except that named semaphores are referenced with a pathname rather than a shared value. Named semaphores are sharable by several processes. Named semaphores have an owner user-id, group-id, and a protection mode.

Unnamed Semaphores are allocated in process memory and initialized. Unnamed semaphores might be usable by more than one process, depending on how the semaphore is allocated and initialized. Unnamed semaphores are either private, inherited through **fork()**, or are protected by access protections of the regular file in which they are allocated and mapped.

Creation:

```
#include <semaphore.h>

sem_t semaphore;
```

Initialisation:

sem_open:

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

The sem_open() function creates a connection between a named semaphore and a process. Once the connection has been created for the semaphore name specified by the *name* argument with a call to sem_open(), the process can use the address returned by the call to reference that semaphore.

name → The name of the semaphore that you want to create or access.

oflag → Flags that affect how the function creates a new semaphore. This argument is a combination of:

- O_CREAT

  This flag is used to create a semaphore if it does not already exist. If O_CREAT is set and the semaphore already exists, then O_CREAT has no effect, except as noted under the description of the **O_EXCL** flag.

- O_EXCL

  If the **O_EXCL** and **O_CREAT** flags are set, the **sem_open** subroutine fails if the semaphore name exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing the **sem_open** subroutine with the **O_EXCL** and **O_CREAT** flags set. If **O_EXCL** is set and **O_CREAT** is not set, **O_EXCL** is ignored.

💡 Don't set *oflag* to O_RDONLY, O_RDWR, or O_WRONLY. A semaphore's behavior is undefined with these flags. The QNX libraries silently ignore these options, but they may reduce your code's portability.

💡 The **O_CREAT** flag requires a third and a fourth parameter: *mode*, which is of type **mode_t**, and *value*, which is of type **unsigned**.

mode_t *mode* → The semaphore's mode (just like file modes). For portability, you should set the read, write, *and* execute bits to the same value. An easy way of doing this is to use the constants from <sys/stat.h>:
• S_IRWXG for group access.
• S_IRWXO for other's access.
• S_IRWXU for your own access.

unsigned int *value* → The initial value of the semaphore. A positive value (i.e. greater than zero) indicates an unlocked semaphore, and a *value* of 0 (zero) indicates a locked semaphore. This value must not exceed SEM_VALUE_MAX.

Return Values:

Upon successful completion, the function returns the address of the semaphore. Otherwise, it will return a value of **SEM_FAILED** and set `errno` to indicate the error. The symbol SEM_FAILED is defined in the header `<semaphore.h>`. No successful return from sem_open() will return the value **SEM_FAILED.**

sem_init:

```
int sem_init(sem_t *semaphore, int pshared, unsigned int value);
```

semaphore → The sem_t pointer to initialize.

pshared → This parameter argument is fundamental in the declaration of semaphore. As it determines the status of the newly initialized semaphore. Whether or not it should be shared between the processes or threads. If the value is non-zero, it means that the semaphore is shared between two or more processes, and if the value is zero, then it means the semaphore is shared between the threads.

value → It specifies the value that is to be assigned to the newly created semaphore that is assigned initially.

Return value:

0 if success.

1 if error.

Blockage:

```
int sem_wait(sem_t *semaphore);
```

Return value:

0 if success.

-1 if error.

Release:

```
int sem_post(sem_t *semaphore);
```

Return value:

0 if success.

-1 if error.

Closure:

```
int sem_close(sem_t *sem);
```

closes the named semaphore referred to bysem, allowing any resources that the system has allocated to the calling process for this semaphore to be freed. ***the semaphore still remains in the system.***

Return value:

0 if success.

-1 if error.

```
int sem_unlink(const char *name);
```

removes the named semaphore referred to by name from the system. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

Return value:

0 if success.

-1 if error.

Good Luck! 🙂