Philosophers

https://en.wikipedia.org/wiki/Dining_philosophers_problem
https://www.youtube.com/watch?v=FY9livorrJI&list=PLfqABt5AS4FmuQf70psXrsMLEDQXNkLq2&index=3

https://www.notion.so/philosophers-VM-c60be9c836084edfbcd9c07e29b429c4
https://github-dotcom.gateway.web.tr/avsrb/Philosophers
https://github.com/nesvoboda/socrates

https://github.com/cacharle/philosophers_test
https://github.com/newlinuxbot/Philosphers-42Project-Tester
https://nafuka11.github.io/philosophers-visualizer/

memset **#include <string.h>**
void *memset(void *b, int c, size_t len);

The memset() function writes len bytes of value c (converted to an unsigned char) to the string b.

printf **#include <stdio.h>**
int printf(cont char *str, ...)

The function printf produces specific output according to a format specifier.

malloc **#include <stdlib.h>**
void *malloc(size_t size);

allocate memory with a given size on the heap.

free **#include <stdlib.h>**
void free(void *ptr);

free previously allocated memory.

write **#include <unistd.h>**

ssize_t write(int fildes, const void *buf, size_t nbyte);

write() attempts to write nbyte of data to the object referenced by the descriptor fildes from the buffer pointed to by buf

usleep **#include <unistd.h>**
int usleep(useconds_t microseconds);

The usleep() function suspends execution of the calling thread until either microseconds
microseconds have elapsed or a signal is delivered to the thread and its action is to invoke a
signal-catching function or to terminate the process. System activity or limitations may
lengthen the sleep by an indeterminate amount.

This function is implemented using nanosleep(2) by pausing for microseconds microseconds or
until a signal occurs. Consequently, in this implementation, sleeping has no effect on the
state of process timers, and there is no special handling for SIGALRM. Also, this implementation
does not put a limit on the value of microseconds (other than that limited by the size of
the useconds_t type); some other platforms require it to be less than one million.

The usleep() function returns the value 0 if successful; otherwise the value -1 is returned

gettimeofday **#include <sys/time.h>**
int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
The system's notion of the current Greenwich time and the current time zone is obtained with the gettimeofday() call.

```
struct timezone {
      int   tz_minuteswest; /* of Greenwich */
      int   tz_dsttime;   /* type of dst correction to apply */
  };

long get_time(void)
{
struct timeval  tp;
long            milliseconds;
```

gettimeofday(&tp, NULL);
milliseconds = tp.tv_sec * 1000;
milliseconds += tp.tv_usec / 1000;
return (milliseconds);
}
https://www.notion.so/philosophers-VM-c60be9c836084edfbcd9c07e29b429c4
https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/

pthread_create #include <pthread.h> ! compile with -pthread or -lpthread ! int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
Threading is a separate set of code execution which runs alongside your main thread.
The pthread_create() function is used to create a new thread, with attributes specified by attr, within a process. If attr is NULL, the
default attributes are used. If the attributes specified by attr are modified later, the thread's attributes are not affected. Upon successful
completion pthread_create() will store the ID of the created thread in the location specified by thread.

6:46

- **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.
- **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

pthread_t ptid;
Threads are sharing memory. Threads share file handlers. the security context and in general system resources are shared.
youtube.com/watch?v=nVESQQg-Oiw
geeksforgeeks.org/thread-functions-in-c-c/

pthread_detach #include <pthread.h>
int pthread_detach(pthread_t thread);
Used to detach a thread. A detached thread does not require a thread to join on

terminating. The resources of the thread are automatically released after terminating if the thread is detached.
This method accepts a mandatory parameter **thread** which is the thread id of the thread that must be detached.

pthread_join **#include <pthread.h>**
int pthread_join(pthread_t thread, void **value_ptr);

Used to wait for the termination of a thread.

- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned in th is stored.

6:46

pthread_mutex_init **#include <pthread.h>**
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
The pthread_mutex_init() function creates a new mutex, with attributes specified with attr. If attr is NULL the default attributes are used.
If successful, pthread_mutex_init() will return zero and put the new mutex id into mutex, otherwise an error number will be returned to indicate the error.

pthread_mutex_destroy **#include <pthread.h>**
int pthread_mutex_destroy(pthread_mutex_t *mutex);
The pthread_mutex_destroy() function frees the resources allocated for mutex.
If successful, pthread_mutex_destroy() will return zero, otherwise an error number will be returned to indicate the error.

pthread_mutex_lock **#include <pthread.h>**
int pthread_mutex_lock(pthread_mutex_t *mutex);
The pthread_mutex_lock() function locks mutex. If the mutex is already locked, the calling thread will block until the mutex becomes available.
If successful, pthread_mutex_lock() will return zero, otherwise an error number will be returned to indicate the error.

pthread_mutex_unlock **#include <pthread.h>**
int pthread_mutex_unlock(pthread_mutex_t *mutex);
If the current thread holds the lock on mutex, then the pthread_mutex_unlock() function unlocks mutex.
Calling pthread_mutex_unlock() with a mutex that the calling thread does not hold will result in undefined behavior.

If successful, pthread_mutex_unlock() will return zero, otherwise an error number will be returned to indicate the error.
The pthread_mutex_unlock() function will fail if the value specified by mutex is invalid or the current thread does not hold a lock on mutex.

/// BONUS ///

fork **#include <unistd.h>**
pid_t fork(void);

Fork system call is used for creating a new process, which is called *child process*, which runs concurrently with the process that makes the fork() call (parent process).
After a new child process is created, both processes will execute the next instruction following the fork() system call.
A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
It takes no parameters and returns an integer value. Below are different values returned by fork().

Negative Value: creation of a child process was unsuccessful.
Zero: Returned to the newly created child process.
Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

kill **#include <signal.h>**
int kill(pid_t *pid*, int *sig*);
The kill() system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.
If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.
If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.
If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is -*pid*.
If *sig* is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

For a process to have permission to send a signal, it must either be privileged (under Linux: have the **CAP_KILL** capability in the user namespace of the target process), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT**, it suffices when the sending and receiving processes belong to the same session. (Historically, the rules were different; see NOTES.)

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

waitpid **#include <sys/wait.h>**
pid_t waitpid(pid_t pid, int *stat_loc, int options);

waitpid can be either blocking or non-blocking:

6:46

- If *options* is 0, then it is blocking
- If *options* is WNOHANG, then is it non-blocking

waitpid is more flexible compared to wait:

- If *pid == -1*, it waits for any child process. In this respect, waitpid is equivalent to wait
- If *pid > 0*, it waits for the child whose process ID equals pid
- If *pid == 0*, it waits for any child whose process group ID equals that of the calling process
- If *pid < -1*, it waits for any child whose process group ID equals that absolute value of pid

sem_open **#include <semaphore.h>** **! compile with -lpthread -lrt!** **sem_t *sem_open(const char *name, int oflag, ...);**
The named semaphore named name is initialized and opened as specified by the argument oflag and a semaphore descriptor is returned to the calling process.

The value of oflag is formed by or'ing the following values:

O_CREAT create the semaphore if it does not exist
O_EXCL error if create and semaphore exists

If O_CREAT is specified, sem_open() requires an additional two arguments. mode specifies the permissions for the semaphore as described in chmod(2) and modi-

fied by the process' umask value (see umask(2)). The semaphore is created with an initial value, which must be less than or equal to SEM_VALUE_MAX.

If O_EXCL is specified and the semaphore exists, sem_open() fails. The check for the existence of the semaphore and the creation of the semaphore are atomic
with respect to all processes calling sem_open() with O_CREAT and O_EXCL set.

When a new semaphore is created, it is given the user ID and group ID which correspond to the effective user and group IDs of the calling process. There is no
visible entry in the file system for the created object in this implementation.

The returned semaphore descriptor is available to the calling process until it is closed with sem_close(), or until the caller exits or execs.

If a process makes repeated calls to sem_open(), with the same name argument, the same descriptor is returned for each successful call, unless sem_unlink()
has been called on the semaphore in the interim.

If sem_open() fails for any reason, it will return a value of SEM_FAILED and sets errno. On success, it returns a semaphore descriptor.

https://www.geeksforgeeks.org/use-posix-semaphores-c/
https://www.geeksforgeeks.org/semaphores-in-process-synchronization/

sem_close **#include <semaphore.h>**
int sem_close(sem_t *sem);

The system resources associated with the named semaphore referenced by sem are deallocated and the descriptor is invalidated.
If successful, sem_close() will return 0. Otherwise, -1 is returned and errno is set.
sem_close() succeeds unless sem is not a valid semaphore descriptor.

sem_post **#include <semaphore.h>**
int sem_post(sem_t *sem);
The semaphore referenced by sem is unlocked, the value of the semaphore is incremented, and all threads which are waiting on the semaphore are awakened.
sem_post() is reentrant with respect to signals and may be called from within a signal hanlder.

If successful, sem_post() will return 0. Otherwise, -1 is returned and errno is set.
sem_post() succeeds unless sem is not a valid semaphore descriptor.

sem_wait **#include <semaphore.h>**
int sem_wait(sem_t *sem);
The semaphore referenced by sem is locked. When calling sem_wait(), if the semaphore's value is zero, the calling thread will block until the lock is acquired or until the call is interrupted by a signal.
If successful (the lock was acquired), sem_wait() will return 0. Otherwise, -1 is returned and errno is set, and the state of the semaphore is unchanged.

sem_unlink **#include <semaphore.h>**
int sem_unlink(const char *name);

The named semaphore named name is removed. If the semaphore is in use by other processes, then name is immediately disassociated with the semaphore, but the
semaphore itself will not be removed until all references to it have been closed. Subsequent calls to sem_open() using name will refer to or create a new semaphore named name.
If successful, sem_unlink() will return 0. Otherwise, -1 is returned and errno is set, and the state of the semaphore is unchanged.