Concurrency Programming Notes

Lab 7,8,9

Threads

- In general, the threads share all resources (such as memory or open files).
- Each thread uses a separate stack, and obviously has its own state of the CPU registers. A limited set of properties might also be specific to a thread (e.g., signal mask, CPU affinity).
- When the main thread returns from `main` function, or when any thread calls `exit` function, all threads are forcefully terminated.
- To compile any program that uses pthreads, one had to add `-pthread` option upon invoking compiler and linker. (except when using a newest compiler)
- Creating a thread:

```
int pthread_create(
        pthread_t * id,
        pthread_attr_t * attr,
        void *(*func)(void *),
        void * arg
);
```

  - It returns 0 on success and something else on failure.
- A thread that has terminated is remembered by the operating system (its return value) until another thread joins it. Alternatively a thread can be detached so that it is not remembered by the OS and cannot be joined.
- Joining a thread:

```
int pthread_join(
        pthread_t thread,
        void ** value_ptr
);
```

  - It waits until a thread has finished and writes it return value to a variable pointed by `value_ptr`. `value_ptr` can be `NULL`, and then the return value is discarded.

- Detaching a thread:

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

  - This function returns without waiting, and the return value of the detached thread will be discarded. A thread can also detach itself.

Mutexes

- To initialize a mutex:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// or
pthread_mutex_t mutex;
int pthread_mutex_init(
        pthread_mutex_t * mutex,
        pthread_mutexattr_t * attr
);
```

- To initialize mutex attributes variable:

```
pthread_mutexattr_t attr;
int pthread_mutextattr_init(pthread_mutexattr_t * attr);
```

- And set its type:

```
int pthread_mutexattr_settype(
        pthread_mutexattr_t * attr,
        int type
);
```

  - where type determines what happens you try to lock a mutex that was locked by the same thread:
    - `PTHREAD_MUTEX_NORMAL` - the thread will deadlock (with itself)
    - `PTHREAD_MUTEX_ERRORCHECK` - locking the mutex will fail (i.e., return `-1` and set `errno` accordingly)
    - `PTHREAD_MUTEX_RECURSIVE` - the mutex counts how many times it was locked, and will unlock after this many unlocks,
    - `PTHREAD_MUTEX_DEFAULT` - it is undefined what will happen.

- To lock a mutex:

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

  - When a mutex is locked it waits until it becomes unlocked.

- To try to lock a mutex:

```
int pthread_mutex_trylock(pthread_mutext_t * mutex);
```

  - When a mutex is locked it returns -1 and sets `errno` to `EBUSY`.

- To unlock a mutex:

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

Critical sections, deadlocks

- Regions of code that must be executed without being interleaved among each other are called critical sections. At most one process may be at a time inside inside a critical section.

- A deadlock is a situation where two or more threads or processes are unable to proceed because each is waiting for the other to release a resource.
- Example:

  > 💡 Imagine two threads, Thread 1 and Thread 2, and two resources, Resource A and Resource B. Each thread needs both resources to complete its task, but they acquire the resources in different orders.

  1. **Thread 1** locks Resource A.
  2. **Thread 2** locks Resource B.
  3. **Thread 1** tries to lock Resource B but must wait because Resource B is already locked by Thread 2.
  4. **Thread 2** tries to lock Resource A but must wait because Resource A is already locked by Thread 1.
- Lock ordering: to prevent deadlocks ensure that all threads acquire locks in a consistent global order. For example, if you have multiple locks (Lock A, Lock B, Lock C), always acquire Lock A before Lock B, and Lock B before Lock C. This prevents circular wait conditions.

Keys

- One can create *keys* (that may be understood as identifiers of variables) and then associate, for each thread separately, a value for a key.
- Each thread can have its own separate instance of data associated with a key, allowing threads to maintain their own context or state independently from other threads. Variables identified by keys cannot be accessed from other threads.
- To create a key:

```
int pthread_key_create(
        pthread_key_t * key,
```

```
        void (*destructor)(void*)
);
```

  - where destructor is a pointer to a function that takes a pointer to a value associated with a key and cleans up the resources.

- To associate a value with it:

```
int pthread_setspecific(
        pthread_key_t key,
        void * value
);
```

  - Each key is initially associated with a `NULL`

- To get associated value:

```
void * pthread_getspecific(pthread_key_t key);
```

Conditional variables

- Conditional variables allow a thread to release a mutex and start waiting until another thread is done with the resources protected by this mutex and signals our thread to lock the mutex again.
- When one thread needs to execute some logic once a specific condition is fulfilled, it should:
  1. lock the mutex
  2. check the condition
     a. if the condition is false wait on the signal (the mutex is unlocked and locked again automatically)
     b. if the condition is true proceed
  3. do the logic
  4. unlock the mutex

- A thread that may change the state and thus affect the condition should:
  1. lock the mutex
  2. do its logic
  3. signal or broadcast the conditional variable
  4. unlock the mutex
- To initialize a condition variable:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- To wait for a conditional variable:

```
int pthread_cond_wait(
        pthread_cond_t * cond,
        pthread_mutex_t * mutex
);
```

- To signal one thread:

```
int pthread_cond_signal(pthread_cond_t * cond);
```

- To broadcast to all threads waiting for the signal:

```
int pthread_cond_broadcast(pthread_cond_t * cond);
```

Lab 10

POSIX

- Types defined by POSIX:
  - `ssize_t` - stores signed size of any data
  - `pid_t` - process identifiers

    - `uid_t` - user identifiers
    - `time_t` - number of seconds
- In POSIX most functions return `-1` on unsuccessful execution and set `errno` accordingly
- To access `errno` we need to `#include <errno.h>`.
- To get human readable explanation of `errno`:
    - `char * strerror(int errnum)` — might not be thread-safe
    - `strerror_r` that requires the programmer to provide a buffer for the message, and `strerror_l` that allows to specify locale(=language); both are thread-safe
    - `void perror(const char * str )` — always uses `errno` to obtain `errnum` and prints `str : explanation` to standard error (or just `explanation` if *str* is `NULL`)

Files

- A program can use multiple files concurrently. Each file is identified by a non-negative integer called a file descriptor.
- POSIX assumes that each newly started program has already three files open. These files are called the standard streams and occupy numbers **0**, **1** and **2**. or, more verbosely, the equivalent fancy constants `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.
- Functions related to files are defined in `unistd.h` and `fcntl.h`.
- To read or write data from a file:

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

    - Where `nbyte` tells how many bytes shall be read/written. Those functions return the number of bytes successfully read/written, unless an error occurred. The number of read/written bytes could be less than `nbyte`.
- Read and write function are guaranteed to execute atomically.

- Reading/writing advances the position in the file.
- In some files (ordinary files, not streams) one may change the position within the file with:

  ```
  off_t lseek(int fd, off_t offset, int whence);
  ```

  - Where `fd` is the file descriptor, `offset` is the offset and whence can be:
    - `SEEK_SET` - offset relative to beginning of the file
    - `SEEK_CUR` - offset relative to the current position
    - `SEEK_END` - offset relative to the end of the file
- Reading from a file at the end of file, or beyond it (set by `lseek`) will simply return 0, not -1.
- To open, or create a file:

  ```
  int open(char * pathname, int flags);
  // To create a file
  int open(char * pathname, int flags, mode_t mode);
  ```

  - Where `mode` defines permissions when creating a file and `flags` must contain either (set by `|` bitwise operator):
    - `O_RDONLY` - for reading only
    - `O_WRONLY` - for writing only
    - `O_RDWR` - for reading and writing
  - `flags` can contain additionally:
    - `O_APPEND` sets file position to the end of a file before every write
    - `O_TRUNC` (shall be used only in conjunction with `O_WRONLY` or `O_RDWR`) truncates the file (sets its size to 0)
    - `O_CREAT` tells `open` that if the file does not exist, it should be created
    - `O_EXCL` (shall be used only in conjunction with `O_CREAT`) makes open fail when the file exists

- To close a file:

```
int close(int fildes);
```

Signals

- Functions related to signals are defined in `signal.h`.
- Signals notify a process about an external event by forcefully switching the control flow of one of the processes threads to a signal handler.
- Signals are told apart by their numbers and corresponding names. For convenience, “the X signal” is written as SIGX.
- The well-defined signals include the following signals:
  - INT - request to interrupt process execution, generated among others by `Ctrl+c`
  - TERM - request to terminate a process
  - KILL - request to kill a process; this is one of two signals that are handled by the operating system and cannot be re-implemented by a programmer
  - STOP - request to stop (pause) a process; the other non-overridable signal
  - CONT - request to continue a stopped process
  - HUP - generated upon a hang-up (shutting down a [pseudo]terminal) — that is, when terminal emulator is closed or a SSH session terminates
- To send a signal:

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

- Processes have default signal handlers, but we can replace those by custom ones.
- A signal handler takes an `int` representing the number of the signal and returns `void`.
- To replace a signal handler:

```
sighandler_t signal(int signum, sighandler_t func);
// with the defined type (defined only here, for readability)
typedef void (*sighandler_t)(int);
```

  - It returns a pointer to a previous signal handler, and takes an `int` representing the signal number, and a pointer to a new signal handler.
  - One can pass pointers to two special signal handlers:
    - `SIG_DFL` - resets the signal handler to default
    - `SIG_IGN` - sets the signal handler to a one that ignores those signals.
- OS stops one of the threads of the process and makes it act as the signal handler was just called in that thread.
- Signal handlers can be executed in any, possibly inconvenient moment.
- Signal handlers can only call some small subset of available functions — those marked with `async-signal-safe` and can only access variables declared outside the signal handler if they are of type `volatile sig_atomic_t`.

Lab 11

Fork

- To work with forks we need `unistd.h`
- To create a new process:

```
pid_t fork();
```

  - It returns the `pid` of the child process in the parent process and 0 in the child process. If it fails it will return -1.
- Fork creates a copy of the running process in particular:
  - all stack and heap is copied
  - the list of open files is copied

    - memory mappings are copied
    - signal handlers are copied
    - pending signals and timers are not copied
    - threads are not copied
    - state of mutexes and values of semaphores are copied
- Because of the last point forking in multithreaded applications in dangerous.
- To get my `pid` and my parents `pid`:

```
pid_t getpid();
pid_t getppid();
```

- A parent process should care for its child processes once they terminate. It might either:
    - Wait for the child to terminate
    - Set up a signal handler for `SIGCHILD` (it suffices to ignore the signal)
- To wait for a termination of any child, or a specific child:

```
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int * stat_loc, int options);
```

    - These functions return the `pid` of terminated process and write the status of the termination to the memory pointed by `stat_loc`.
    - In `waitpid` the `pid` can also be negative, in particular when it is -1 then it waits for any child
    - `options` is a combinations of flags, for `options` equal to 0 it simply waits for a child to terminate, but for example the flag `WNOHANG` check if the child already has terminated.

Exec

- Exec family of functions ask the OS to replace all the memory of the current program with the code and data of the specified program and execute it.
- Exec doesn't create a new process, one has to manually use fork for that.
- All exec family functions (defined in `unistd.h`):

```
int execlp(char * file, char * arg0, /*,..., NULL*/);
int execl(char * path, char * arg0, /*,..., NULL*/);
int execle(char * path, char * arg0, /*,..., NULL*/, char * e
int execvp(char * file, char * argv[]);
int execv(char * path, char * argv[]);
int execve(char * path, char * argv[], char * envp[]);
```

  - It returns -1 on failure, however it is pointless to check the return value since if it were successful then the next instructions after exec wouldn't execute.
  - functions ending with l... take a list of arguments whereas functions ending with v... take a pointer to a vector of arguments (in both cases the last argument should be `NULL` and the first the program name).
  - functions ending with p take a file name and search for the right program, and functions not ending with p require a path to the executable.
  - functions ending with e take an array of environmental variables (ending with `NULL`) that should be used instead of the ones inherited from our program.
- Calling exec retains the list of open files, however almost all other resources are released before executing the new program.
- `getenv` and `setenv` get and change environmental variables.

Duplicating file descriptors

- To duplicate file descriptors we need `unistd.h`
- To duplicate file descriptors:

```
int dup(int fildes);
int dup2(int fildes, int target);
```

- Duplicating file descriptors differs from opening the same file twice, because all the flags are copied from the first descriptor automatically and advancing the position in a file with one descriptor does the same with the other.
- `dup2` atomically closes file descriptor `target` and then duplicates the descriptor `fildes` to the descriptor `target`. This is used to change where for example the standard output goes, we can provide the descriptor of the std output as `target` and a descriptor to our new file as `fildes`.
- After duplicating each file descriptor has to be closed separately.

Blocking vs non-blocking operations

- Some functions need to wait for something to complete before proceeding, like for example waiting for a mutex to unlock or waiting for user input. This is called blocking. Functions that may block are called blocking.
- Blocking might take arbitrary amount of time.
- Usually there is a way to call blocking functions in a non-blocking mode. When we do so then this function either does what it's supposed to do without waiting or it returns -1.
- For functions related to file descriptors the blocking/non-blocking mode is selected by the `O_NONBLOCK` flag for the file descriptor.
- To set (or clear this flag we need to:

```
flags = fcntl(fd, F_GETFL);
flags |= O_NONBLOCK;
// flasg &= ~O_NONBLOCK; for clearing
fcntl(fd, F_SETFL, flags);
```

Pipes & FIFOs

- For pipes we need to include `unistd.h`

- Pipes are an unidirectional communication channel used to send data from one process to another (or one thread to another).
- Internally, a pipe is just a pair of file descriptors, one for reading, one for writing.
- To create a pipe:

```
int pipe(int fildes[2]);
```

  - where `pipe` opens `fildes[0]` reading and `fildes[1]` for writing.
- By default pipes are blocking, reading or writing may need waiting.
- When all file descriptors that allowed writing to a pipe are closed, a read from the pipe will return 0.
- When all file descriptors that allowed reading from a pipe are closed, a write to the pipe will first raise SIGPIPE, then return -1 and set `errno` to `EPIPE`.
- To share a pipe between two processes, one must create a pipe in one process and `fork` – file descriptors are copied upon forking.
- A **FIFO** file (or a named pipe) is a special file that allows opening either end of a pipe by providing a path to the file. A FIFO file can be created with `mkfifo` shell utility or by the `mkfifo` function.
- A call to `open` on a path to a FIFO file is blocking. `open` returns only once at least one process invoked `open` with `O_RDONLY` and at least one process invoked `open` with `O_WRONLY`). From that point on the file descriptors act as those of an (anonymous) pipe.

Lab 12

Shared memory

- It is possible for two processes to use the same physical address range of the main memory. Such memory is called shared memory.

- Keep in mind that although two processes use the same physical addresses, the virtual addresses are usually different.
- So do not store any pointers in the shared memory - they will not be valid for other processes sharing the memory. A pointer points to a location is virtual address space which is still specific to this particular process even if we use shared memory. Storing offsets (differences between addresses) within a continuous shared memory address range is fine.
- There is an exception to the two paragraphs above — when two processes share memory as a result of a `fork`, then the addresses naturally match (warning! this probably only works right after `fork`, couldn't find much info on it)
- To use shared memory we need to ask OS to set up a new (possibly shared) memory address range with this function (we need to include `sys/mman.h` before):

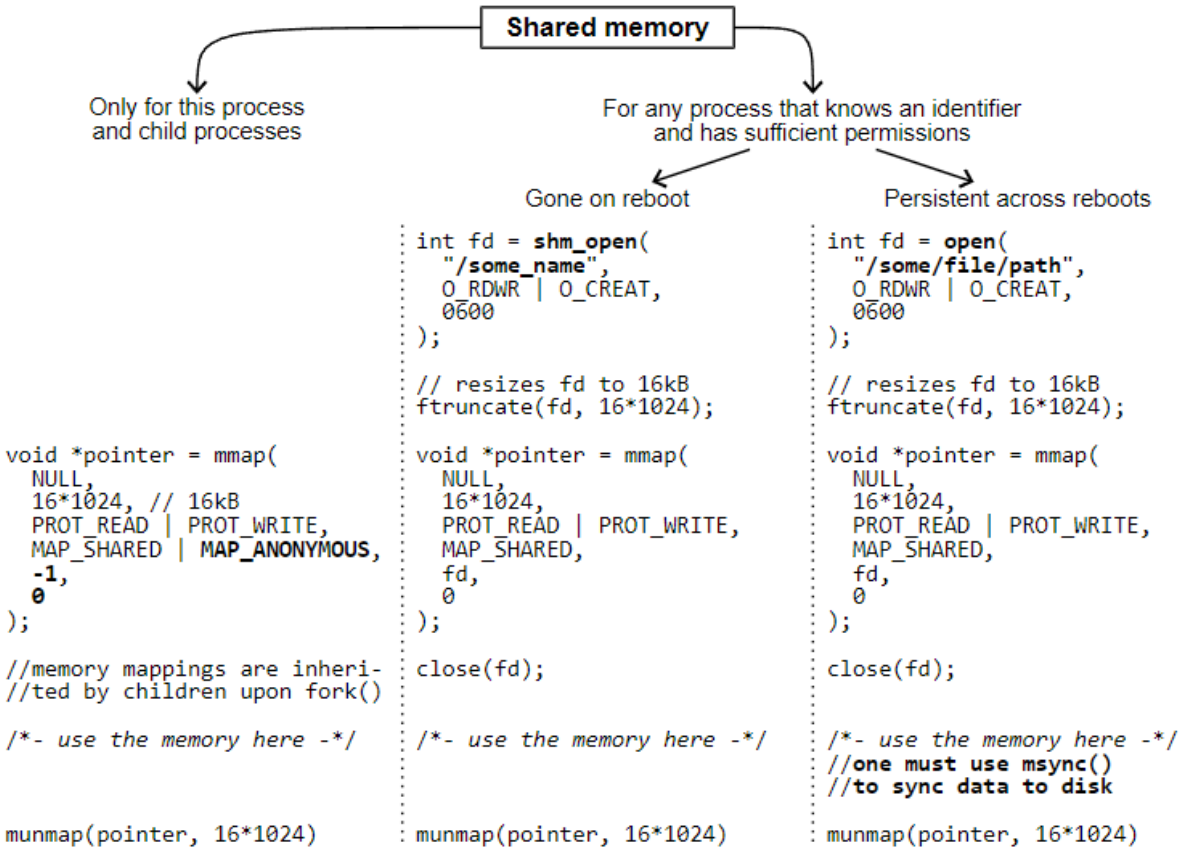
```
void * mmap(
        void * addr,
        size_t len,
        int prot,
        int flags,
        int fildes,
        off_t off
);
```

  - with `flags` including the `MAP_SHARED` flag.
  - It returns an address to the start of the newly created address range.
  - `mmap` is used to either copy a files contents into memory (for efficiency) or just allocate some new (possibly shared) memory.
  - To use `mmap` to just allocate some memory (which can only be shared among this process and its child processes) we need to add `MAP_ANONYMOUS` flag, set `fildes` to -1 and `off` to 0.
  - To use `mmap` to copy an ordinary file into memory we need to set `fildes` to this file's descriptor and OS automatically copies its contents into memory.

To write back the changes from the memory to the file we need to do it manually with `msync`.

- `fildes` can also refer to a shared memory object. We can get such file descriptor with `shm_open` which has identical arguments to `open`.

- To use `mmap` we need to provide the size of the file. If our file is smaller than the size we provided then `mmap` succeeds, but we have to be careful not to access memory beyond the file.
- To change files size we can use:
  - `ftruncate` to truncate a file (from `unistd.h`)
  - `posix_fallocate` to ensure that a file is at least of a given size (from `fcntl.h`)
- To clean up memory mapping we can use `munmap`.
- `shm_open`, `mmap`, `msync` and `munmap` are defined in `sys/mman.h`.



- To compile programs that use `shm_open` with older `glibc` versions, one must add `-lrt` to compile options.

Semaphores

- A semaphore is a mechanism to control concurrency.
- Semaphores are used in scenarios when you need to limit the number of threads that can access a particular resource. For example, if you have a pool of connections to a database and you only want up to 10 threads to access the pool simultaneously, you can use a counting semaphore initialized to 10.

Library analogy [edit]

Suppose a physical library has ten identical study rooms, to be used by one student at a time. Students must request a room from the front desk. If no rooms are free, students wait at the desk until someone relinquishes a room. When a student has finished using a room, the student must return to the desk and indicate that the room is free.

In the simplest implementation, the clerk at the front desk knows only the number of free rooms available. This requires that all of the students use their room while they have signed up for it and return it when they are done. When a student requests a room, the clerk decreases this number. When a student releases a room, the clerk increases this number. The room can be used for as long as desired, and so it is not possible to book rooms ahead of time.

In this scenario, the front desk count-holder represents a counting semaphore, the rooms are the resource, and the students represent processes/threads. The value of the semaphore in this scenario is initially 10, with all rooms empty. When a student requests a room, they are granted access, and the value of the semaphore is changed to 9. After the next student comes, it drops to 8, then 7, and so on. If someone requests a room and the current value of the semaphore is 0,[2] they are forced to wait until a room is freed (when the count is increased from 0). If one of the rooms was released, but there are several students waiting, then any method can be used to select the one who will occupy the room (like FIFO or randomly picking one). And of course, a student must inform the clerk about releasing their room only after really leaving it.

- Semaphores are either considered *binary* or *counting*: the former may have value of 0 or 1, and the latter may have any non-negative value.
- Decrementing a semaphore which value is zero waits until the value becomes nonzero, that is until another process / thread increments the semaphore.
- A semaphore may be increased by a process which hasn't decreased it's value before, therefore a semaphore can have its value greater than the initial value.

- Increasing a binary semaphore will have no effect.
- To create a semaphore (needs `semaphore.h` and `fcntl.h`):

```
// Named semaphore
sem_t * sem_open(char * name, int oflag);
sem_t * sem_open(char * name, int oflag, mode_t mode, unsigne
// Unnamed semaphore
int sem_init(sem_t * sem, int pshared, unsigned value);
```

  - Where `value` is the initial value of the semaphore and `pshared` sets whether the semaphore should work for threads of distinct processes or not.
- To increment/decrement a semaphore:

```
// Increment
int sem_post(sem_t * sem);
// Decrement (possibly waiting)
int sem_wait(sem_t * sem);
```

- To get the current value of a semaphore:

```
int sem_getvalue(sem_t * sem, int *sval);
```

- Once an unnamed semaphore is no longer needed by any process, it shall be destroyed by `sem_destroy`.
- Once an named semaphore is no longer needed by this process, it shall be closed with `sem_close`.
- Once the name of a named semaphore is no longer needed, it shall be removed by `sem_unlink`.
- To compile programs that use semaphores with older `glibc` versions, one must add `-pthread` to compile options.

Differences between mutexes and semaphores

- A mutex can only be unlocked by the thread that locked it

- A recursive mutex has to be unlocked the exact number of times it was locked and a semaphore only needs to be unlocked once.
- A semaphore (binary or counting) can be increased however many times you want (in case of binary it will just be ignored), but a mutex cannot be unlocked if it hasn’t been already locked.