Lab Notes

Lab 2 Ideas

Pretty irrelevant from a compiling and writing code point of view, i.e., don't know how important they are for the lab exam:

Source Files and Compilation

- The source files conventionally have the extension `.c`.
- To compile the program `name.c`, run:

  ```
  gcc -o name name.c
  ```

- The name argument of the `-o` option indicates the name given to the executable. If `-o name` is not used, the executable is named `a.out`.
- In UNIX, executables are identified by execution permission, not by extension (e.g. `.exe`).
- To compile programs with standard math functions, include the `libm` library:

  ```
  gcc -lm -o prog prog.c
  ```

- For threads, include the `libpthread` library:

  ```
  gcc -lpthread -o prog prog.c
  ```

- Enable warnings with `-wall`.
- The options can be entered in any order. Note: `-o name` is a single option and must not be split.

Command Line Details

- Example command:

```
./myecho <space> arg1 <space> arg2 <space> arg3 <space> done
```

- Explanation:
  - `./`: Represents the current directory and the path separator.
  - `myecho`: The executable file.
  - `<space>`: Spaces separate command-line elements.
  - Arguments (`arg1`, `arg2`, etc.) are passed to the program.
- Be cautious with spaces, as they can unintentionally break elements. Use quotation for arguments with spaces.

Using `getopt`

- Use `getopt` from `unistd.h` to process arguments.
- Key elements:
  - `int opterr`: Controls error message behavior.
  - `int optopt`: Stores invalid options.
  - `int optind`: Index of the next `argv` element.
  - `char *optarg`: Pointer to the option's argument.
- Function prototype:

```
int getopt(int argc, char **argv, const char *options);
```

- Example options:
  - `':'`: Indicates a required argument.
  - `'::'`: Indicates an optional argument (GNU extension).
- The function returns `-1` when the processing is complete. Use `optind` to access remaining arguments.

Lab 3 Ideas - I have no idea how to read and write

Pretty important concepts, especially since you have the biggest problems here.

File Descriptors and Handles

- Files are accessed through a handle (file descriptor), an integer associated with the file when opened.
- Maximum file descriptors per process: 32,767. Constraints may vary based on system resources.

Opening Files

- Use the `open` function:

```
int open(const char *pathname, int oflag, [, mode_t mode]);
```

- Parameters:
  - `pathname`: The file name.
  - `oflag`: Flags specifying file access mode.
  - `mode`: Permissions (required with `O_CREAT`).
- Common flags:

```
O_RDONLY - open for read only
O_WRONLY - open for write only
O_RDWR - open for read and write
O_APPEND - append to the file
O_CREAT - create the file if it does not exist
O_EXCL - exclusive creation (error if file exists)
O_TRUNC - truncate existing file
```

File Permissions

- Numeric permissions use a three-digit octal value. Each digit represents:
  - Owner permissions.
  - Group permissions.
  - Others' permissions.
- Permission values:

  ```
  r (read) - 4
  w (write) - 2
  x (execute) - 1
  ```

- Example: `744`
  - Owner: `rwx = 7 (4+2+1)`
  - Group: `r-- = 4 (4+0+0)`
  - Others: `r-- = 4 (4+0+0)`

We can use flags to give permissions to files. The `mode` parameter is used only when the file is created and specifies the access rights associated with the file. These constants can be combined using the OR operator (`|`):

- `S_IRUSR`: Read permission for the file owner (user).
- `S_IWUSR`: Write permission for the file owner (user).
- `S_IXUSR`: Execution permission for the file owner (user).
- `S_IRGRP`: Read permission for the group that owns the file.
- `S_IWGRP`: Write permission for the group that owns the file.
- `S_IXGRP`: Execution permission for the group that owns the file.
- `S_IROTH`: Read permission for other users.
- `S_IWOTH`: Write permission for other users.
- `S_IXOTH`: Execution permission for other users.

Creating Files

Files can be created using the following function:

```
int creat(const char *pathname, mode_t mode);
```

This is equivalent to specifying the `O_WRONLY | O_CREAT | O_TRUNC` options to the `open` function.

Opening and Closing Files

Files must be closed after use to avoid unexpected errors:

```
int close(int filedes);
```

Where `filedes` is the file descriptor obtained from `open`. Avoid closing a file that hasn't been opened or has already been closed.

Read and Write Functions

- Reading data:

  ```
  ssize_t read(int fd, void *buff, size_t nbytes);
  ```

  Reads up to `nbytes` from the file descriptor `fd` into the buffer pointed to by `buff`. Returns the number of bytes read or `-1` on error.

- Writing data:

  ```
  ssize_t write(int fd, void *buff, size_t nbytes);
  ```

  Writes `nbytes` from the buffer pointed to by `buff` to the file descriptor `fd`. Returns the number of bytes written or `-1` on error.

Positioning in Files

The `lseek` function adjusts the current position in the file:

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

Where:

- `SEEK_SET`: Position relative to the beginning of the file.

- SEEK_CUR: Position relative to the current position.
- SEEK_END: Position relative to the end of the file.

Returns -1 on error.

Directory Operations

- Create a directory:

```
int mkdir(const char *pathname, mode_t mode);
```

- Remove a directory:

```
int rmdir(const char *pathname);
```

Library Functions for Files

- Open a file:

```
FILE *fopen(const char *filename, const char *mode);
```

- Close a file:

```
int fclose(FILE *stream);
```

- Write to a file with formatting:

```
int fprintf(FILE *stream, const char *format, ...);
```

- Read from a file:

```
int fscanf(FILE *stream, const char *format, ...);
```

- Binary read/write:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Directory Reading

Directories can be read using the following functions:

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
int closedir(DIR *dir);
```

The `dirent` structure contains:

```
char d_name[];
```

This field holds the name of the directory entry. Use library functions for better portability.

File Links and Paths

Additional functions include:

- Create a hard link:

  ```
  int link(const char *oldpath, const char *newpath);
  ```

- Create a symbolic link:

  ```
  int symlink(const char *oldpath, const char *newpath);
  ```

- Delete a directory entry:

  ```
  int unlink(const char *pathname);
  ```

- Rename/move files:

  ```
  int rename(const char *oldpath, const char *newpath);
  ```

- Change the current directory:

  ```
  int chdir(const char *path);
  ```

- Determine the current directory:

  ```
  char *getcwd(char *buf, size_t size);
  ```

Lab 4 Ideas - The Fun Begins

A process can be in one of three states at any given time:

- **In execution:** Actively running on the CPU.
- **Ready for execution:** Waiting to be scheduled by the operating system.
- **Blocked:** Suspended due to a time-consuming operation (e.g., input/output) or intentional suspension.

The `fork()` System Call

Processes in UNIX are created using the `fork()` system call:

```
pid_t fork();
```

Key Points:

- The calling process is the *parent*, and the created process is the *child*.
- The `fork()` call creates a new process that is a faithful copy of the parent.
- Both the parent and child have independent stacks, data areas, and program counters.
- The child inherits file descriptors opened by the parent.
- The return value of `fork()` differentiates the two processes:
  - `-1`: An error occurred.
  - `0`: Returned to the child process.
  - `pid`: Returned to the parent, where `pid` is the child's process ID.

`wait()` and `waitpid()` Functions

The `wait()` and `waitpid()` functions are used to retrieve the status of a terminated child process:

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int flags);
```

Key Points:

- `wait()` waits for any child of the current process.
- `waitpid()` waits for a specific child process, identified by `pid`.
- The `status` parameter allows retrieval of the child's return value using macros.

Functions Like `exec()`

The `exec()` family of functions replaces the current process image with a new program:

```
execl(), execlp(), execv(), execvp(), execle(), execve()
```

Key Points:

- These functions launch a new program from disk, replacing the current code, data, and stack.
- The process ID (`PID`) remains unchanged.
- If the operation fails, the function returns `-1`.
- Typical usage:
  - Call `fork()` to create a new process.
  - In the child process, call `exec()` to replace the program.

`system()` and `vfork()` Functions

- `system()` launches a program from disk, combining `fork()`, `exec()`, and `waitpid()`:

  ```
  int system(const char *cmd);
  ```

- `vfork()` is similar to `fork()`, but it does not copy the parent's address space:

```
pid_t vfork();
```

- Used with `exec()`, it saves time by skipping unnecessary copying.

Other Process Functions

- `pid_t getpid()`: Returns the PID of the current process.
- `pid_t getppid()`: Returns the PID of the parent process.
- `uid_t getuid()`: Returns the user ID of the process owner.
- `gid_t getgid()`: Returns the group ID of the process owner.

Lab 5 - He said, "Damn, Nicki, it's tight," I-I say, "Yeah, baby, you right" He said, "Damn, bae, you so little, but you be really takin' that pipe"

Introduction to Pipes

A **pipe** in UNIX is a mechanism for interprocess communication that connects two related processes (e.g., ancestor-descendant relationship). It allows unidirectional communication, with one end used for writing and the other for reading.

Creating Pipes

To create a pipe, the `pipe()` system call is used:

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

Key Points:

- `filedes[0]`: File descriptor for the read end of the pipe.
- `filedes[1]`: File descriptor for the write end of the pipe.
- Returns `0` on success and `-1` on error.

Steps for Process Communication via Pipes

1. The parent process creates a pipe using `pipe()`.
2. The parent process calls `fork()` to create a child process.
3. The child process:
   - Closes the read end (`filedes[0]`).
   - Writes data to the pipe using `write()`.
4. The parent process:
   - Closes the write end (`filedes[1]`).
   - Reads data from the pipe using `read()`.

Pipe Behavior

- Pipes act like a queue: writes insert data, and reads remove data.
- Writing is blocked if the pipe is full (minimum capacity: 512 bytes as per POSIX).
- Reading returns `0` if all writers have closed the write end and there is no more data.
- Writing generates a `SIGPIPE` signal if all readers have closed the read end, causing an error.

Duplication and Redirection

Duplication and redirection are performed using:

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- `dup()` duplicates `oldfd` and returns the smallest available file descriptor.
- `dup2()` duplicates `oldfd` to `newfd`. If `newfd` is already open, it is first closed.
- Both functions return the new descriptor or `-1` on error.

Example Code: Pipe Communication

The following example demonstrates communication between a parent and child process using a pipe:

```
if (pid == 0) /* Child process */
{
    close(pfd[0]); /* Close the reading end */
    dup2(pfd[1], 1); /* Redirect standard output to the pipe */
    execlp("ls", "ls", "-l", NULL); /* Run the ls command */
    printf("Error at exec\n"); /* Exec returned, indicating failure */
}
else /* Parent process */
{
    close(pfd[1]); /* Close the writing end */
    stream = fdopen(pfd[0], "r"); /* Open a stream for the reading end */
    fscanf(stream, "%s", string); /* Read from the pipe */
    close(pfd[0]); /* Close the used end */
    exit(0);
}
```

Note: Always close unused pipe ends. Each pipe should be used for one communication direction.

Observations

1. The number of bytes a pipe can hold without being read is implementation-dependent. Minimum capacity is 512 bytes (POSIX).

2. The `read()` function returns `0` when all writers have closed their write ends and the pipe is empty.

3. Writing to a pipe with no readers generates a `SIGPIPE` signal and returns an error.

4. Writes to a pipe are atomic only if fewer bytes than `PIPE_BUF` are written.

Lab 6 - Dam semnale cu fum sa ne vezi de departe

Introduction to Signals

Signals are a way of expressing asynchronous events in the system. A process can handle signals in three ways:

- Capture and execute an action using a signal handler.
- Ignore the signal.
- Execute the default action, which could terminate or ignore the signal.

Common Signals

POSIX.1 defines several signals, each associated with a specific action. The following table summarizes the common signals:

Signal	**Number**	**Description**
SIGHUP	1	Hangup - terminal used by the process was closed.
SIGINT	2	Interrupt - e.g., pressing Ctrl-C to terminate a process.
SIGQUIT	3	Quit - user requests program termination.
SIGILL	4	Illegal Instruction - invalid opcode or insufficient privileges.
SIGABRT	6	Abort - abnormal termination via `abort(3)`.
SIGFPE	8	Floating Point Exception - e.g., division by zero.
SIGKILL	9	Kill - immediately terminates the process. Cannot be caught or ignored.
SIGUSR1, SIGUSR2	10, 12	User-defined signals for custom actions.
SIGSEGV	11	Segmentation Fault - invalid memory reference.
SIGPIPE	13	Broken Pipe - write to a pipe with no readers.
SIGALRM	14	Alarm Timer - signal after `alarm()` expires.
SIGTERM	15	Terminate - normal termination request.
SIGCONT	18	Continue - resumes a process stopped by SIGSTOP.
SIGSTOP	19	Stop - suspends the process until SIGCONT is received. Cannot be caught or ignored.

`signal()` System Call

The `signal()` system call is used to capture signals for processing:

```
#include <signal.h>
typedef void (*sighthandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Parameters:

- `signum`: Signal number to handle.
- `handler`: Function to handle the signal, or special values:
  - `SIG_IGN`: Ignore the signal.
  - `SIG_DFL`: Reset to default behavior.

Returns the previous signal handler or `SIG_ERR` on error. Signals like `SIGKILL` and `SIGSTOP` cannot be caught or ignored.

`sigaction()` System Call

The `sigaction()` system call provides more control over signal handling:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *
    oldact);
```

Structure:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

`kill()` System Call

The `kill()` system call sends signals to specific processes:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Effect Based on `pid`:

- `pid > 0`: Signal sent to process with the given PID.
- `pid == 0`: Signal sent to all processes in the same process group.
- `pid == -1`: Signal sent to all processes (except `init` and the sender).
- `pid < -1`: Signal sent to processes in the group `-pid`.

`raise()` and `abort()` Functions

- `raise()` sends a signal to the current process:

  ```
  #include <signal.h>
  int raise(int sig);
  ```

- `abort()` sends a `SIGABRT` signal to the current process:

  ```
  #include <stdlib.h>
  void abort(void);
  ```

  If `SIGABRT` is blocked or ignored, `abort()` still terminates the process abnormally.

`alarm()` Function

The `alarm()` function sends a `SIGALRM` signal after a specified time:

```
#include <unistd.h>
unsigned int alarm(int seconds);
```

Behavior:

- Cancels any previous alarm if `seconds == 0`.
- Returns the remaining seconds of the previous alarm.

Observations and Recommendations

- Handlers for signals should be defined before forking processes to prevent race conditions.
- Use `SIG_IGN` to ignore signals temporarily during setup.
- Be cautious with signals like `SIGINT`, as they can terminate processes prematurely.
- Always check the result of system calls like `fork()`, `kill()`, and `alarm()` for errors.

1 Lab 7: Threads

A thread can be seen as a stream of instructions that executes. At the operating system level, parallel execution of threads is achieved in a similar way to that of processes, by switching between threads according to a scheduling algorithm. Unlike the case of processes, however, the switching can be done much faster because the information stored by the system for each thread is much less than in the case of processes. Threads have very few resources of their own. Essentially, a thread can be seen as a program counter, a stack, and a set of registers, while all other resources (e.g., data area, file identifiers) belong to the process in which it runs and are exploited in common.

Linux implements threads by providing, at a low level, the `clone()` system call:

```
pid_t clone(void * sp , unsigned long flags )
```

The `clone()` function is an alternative interface to the `fork()` system function, which creates a child process but offers more options for creation. If `sp` is nonzero, the child process will use `sp` as its stack pointer, thus allowing the programmer to choose the stack for the new process. The `flags` argument is a string of bits containing various options for creating the child process. The lower byte of `flags` contains the signal that will be sent to the parent when the newly created child terminates. Other options for `flags` include:

- `COPYVM`: If set, the child's memory pages will be exact copies of the parent's memory pages (similar to `fork()`). If not set, the child will share its memory pages with the parent.

- `COPYFD`: If set, the child will receive the parent's file descriptors as distinct copies. If not set, the child will share the file descriptors with the parent.

The function returns the PID of the child in the parent and zero in the child. Therefore, the `fork()` system call is equivalent to:

```
clone(0, SIGCLD | COPYVM)
```

The `clone()` function offers sufficient facilities to create thread-type primitives. However, it is advisable not to use `clone()` directly for two reasons:

- It is not portable, being Linux-specific.
- Its usage is cumbersome.

Modern Linux distributions provide the LinuxThreads library, which adheres to the POSIX standard. To use threads, programs should be compiled using:

```
gcc -D_REENTRANT -lpthread <file.c> -o <executable>
```

It is necessary to define the `_REENTRANT` constant for reasons related to parallel execution of threads and explicitly include the `pthread` library.

1.1 Creating Threads

Threads are created using the `pthread_create()` function:

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

The function creates a thread that executes in parallel with the creator thread. The new thread will execute the `start_routine` function, which must have a single argument of type `void *`. The `arg` parameter is the argument passed to this function. The `attr` parameter specifies options for creating the thread (usually given as `NULL` to accept default options). The thread identifier is returned in the variable indicated by the `thread` parameter. The function returns 0 if creation was successful and a non-zero value otherwise.

Threads terminate either by explicitly calling the `pthread_exit()` function or implicitly by exiting the `start_routine` function.

1.2 Terminating Threads

A thread can be terminated by calling:

```
void pthread_exit(void *retval);
```

The `retval` value is the value returned by the thread upon termination. The status of a thread can be retrieved using:

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

This function blocks the calling thread until the thread identified by `th` terminates. The state of the terminated thread is returned via the `thread_return` parameter.

Threads are categorized as:

- **Joinable**: Their states can be retrieved by other threads using `pthread_join()`.
- **Detached**: Their states cannot be retrieved, and resources are fully deallocated upon termination.

A joinable thread can be made detached using the `pthread_detach()` function.

1.3 Observations

- A process consists of a single thread upon creation, known as the main (initial) thread.
- All threads within a process execute in parallel and share the same data area. Thus, global variables are shared among threads.
- If a process terminates due to receiving a signal, all its threads terminate as well.
- Calling `exit()` from any thread will terminate the entire process and all threads within it.
- Any function or system call affecting processes impacts the entire process, regardless of the calling thread. For example, `sleep()` will suspend all threads.

Lab 8: Mutexes and Fuck Semaphores

1.4 Introduction to Race Conditions and Critical Sections

Situations where the outcome of a system consisting of multiple processes (or threads) depends on their relative execution speed are called **race conditions**. Race conditions occur when shared resources are accessed without ensuring mutual exclusion. Shared resources can include shared memory, files, or database records.

To solve the race condition problem, the concept of a **critical section** is introduced. A critical section is a part of the program that accesses shared resources. Ensuring that no two threads execute in their critical sections simultaneously is called **mutual exclusion**.

1.5 Synchronization Mechanisms: Mutexes and Semaphores

Mutexes and semaphores are two synchronization mechanisms used to enforce mutual exclusion and coordinate the execution of threads.

1.5.1 Mutexes in C

A **mutex** (short for mutual exclusion) is a locking mechanism that allows only one thread to access a shared resource at a time. A mutex is initialized, locked before entering the critical section, and unlocked after leaving the critical section.

Example: Using Mutex in C The following example demonstrates how to use a mutex to protect a shared variable:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

// Shared variable
int shared_variable = 0;

// Mutex declaration
```

```
pthread_mutex_t lock;

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        // Lock the mutex before entering the critical section
        pthread_mutex_lock(&lock);
        shared_variable++;
        // Unlock the mutex after leaving the critical section
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Initialize the mutex
    pthread_mutex_init(&lock, NULL);

    // Create two threads
    pthread_create(&thread1, NULL, increment, NULL);
    pthread_create(&thread2, NULL, increment, NULL);

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Destroy the mutex
    pthread_mutex_destroy(&lock);

    printf("Final value of shared_variable: %d\n", shared_variable);
    return 0;
}
```

Explanation:

- `pthread_mutex_init`: Initializes the mutex.

- `pthread_mutex_lock`: Locks the mutex to ensure no other thread can enter the critical section.
- `pthread_mutex_unlock`: Unlocks the mutex after the critical section is completed.
- `pthread_mutex_destroy`: Cleans up resources used by the mutex.

1.5.2 Semaphores in C

A **semaphore** is a synchronization primitive that can be used for mutual exclusion or signaling between threads. Semaphores can be of two types:

- **Binary Semaphore:** Takes values 0 or 1, used for mutual exclusion.
- **Counting Semaphore:** Takes values greater than or equal to 0, used for signaling.

Semaphores in C are implemented using the `<semaphore.h>` library.

Example: Using Semaphores in C The following example demonstrates how to use a semaphore to synchronize access to a shared resource:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

// Shared variable
int shared_variable = 0;

// Semaphore declaration
sem_t semaphore;

void *increment(void *arg) {
    for (int i = 0; i < 1000000; i++) {
        // Wait (P operation) before entering the critical section
        sem_wait(&semaphore);
        shared_variable++;
        // Signal (V operation) after leaving the critical section
        sem_post(&semaphore);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // Initialize the semaphore with a value of 1 (binary semaphore)
    sem_init(&semaphore, 0, 1);

    // Create two threads
    pthread_create(&thread1, NULL, increment, NULL);
    pthread_create(&thread2, NULL, increment, NULL);

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Destroy the semaphore
    sem_destroy(&semaphore);

    printf("Final value of shared_variable: %d\n", shared_variable);
    return 0;
}
```

Explanation:

- `sem_init`: Initializes the semaphore. The second argument specifies whether the semaphore is shared between threads (`0`) or processes (`non-zero`).
- `sem_wait`: Decrements the semaphore value. If the value is 0, the thread is blocked until the value becomes positive.
- `sem_post`: Increments the semaphore value, unblocking a waiting thread if necessary.
- `sem_destroy`: Cleans up resources used by the semaphore.

1.6 Comparison of Mutexes and Semaphores

- **Mutex:** Designed specifically for mutual exclusion. Only one thread can lock a mutex at a time, and the thread that locks it must unlock it.
- **Semaphore:** Can be used for both mutual exclusion (binary semaphore) and signaling (counting semaphore). Any thread can signal or wait on a semaphore.

When to Use:

- Use **mutexes** when you need simple locking for mutual exclusion.
- Use **semaphores** when you need more complex signaling or coordination between threads.