Section 2: Processes, Threads, Locks

CS 162

September 11-13, 2019

Contents

1 Vocabulary

- **fork** - A C function that calls the fork syscall that creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process (except for a few details, read more in the man page). Both the newly created process and the parent process return from the call to fork. On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.

- **wait** - A class of C functions that call syscalls that are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

- **pipe** - A system call that can be used for interprocess communication.

  More specifically, the `pipe()` syscall creates two file descriptors, which the process can `write()` to and `read()` from. Since these file descriptors are preserved across `fork()` calls, they can be used to implement inter-process communication.

```
/* On error, pipe() returns -1. On success, it returns 0
 * and populates the given array with two file descriptors:
 *  - fildes[0] will be used to read from the data queue.
 *  - fildes[1] will be used to write to the data queue.
 *
 * Note that whether you can write to fildes[0] or read from
 * fildes[1] is undefined. */
int pipe(int fildes[2]);
```

- **exit code** - The exit status or return code of a process is a 1 byte number passed from a child process (or callee) to a parent process (or caller) when it has finished executing a specific procedure or delegated task

- **exec** - The exec() family of functions replaces the current process image with a new process image. The initial argument for these functions is the name of a file that is to be executed.

- **pthreads** - A POSIX-compliant (standard specified by IEEE) implementation of threads. A `pthread_t` is usually just an alias for "`unsigned long int`".

- **pthread_create** - Creates and immediately starts a child thread running in the same address space of the thread that spawned it. The child executes starting from the function specified. Internally, this is implemented by calling the clone syscall.

```
/* On success, pthread_create() returns 0; on error, it returns an error
 * number, and the contents of *thread are undefined. */
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- **pthread_join** - Waits for a specific thread to terminate, similar to `waitpid(3)`.

  ```
  /* On success, pthread_join() returns 0; on error, it returns an error number. */
  int pthread_join(pthread_t thread, void **retval);
  ```

- **pthread_yield** - Equivalent to thread_yield() in Pintos. Causes the calling thread to vacate the CPU and go back into the ready queue without blocking. The calling thread is able to be scheduled again immediately. This is not the same as an interrupt and will succeed in Pintos even if interrupts are disabled.

  ```
  /* On success, pthread_yield() returns 0; on error, it returns an error number. */
  int pthread_yield(void);
  ```

- **critical section** - A section of code that accesses a shared resource and must not be concurrently run by more than a single thread.
- **race condition** - A situation whose outcome is dependent on the sequence of execution of multiple threads running simultaneously.
- **lock** - Synchronization primitives that provide mutual exclusion. Threads may acquire or release a lock. Only one thread may hold a lock at a time. If a thread attempts to acquire a lock that is held by some other thread, it will block at that line of code until the lock is released and it successfully acquires it. Implementations can vary.
- **semaphore** - Synchronization primitives that are used to control access to a shared variable in a more general way than locks. A semaphore is simply an integer with restrictions on how it can be modified:
  - When a semaphore is initialized, the integer is set to a specified starting value.
  - A thread can call `down()` (also known as **P**) to attempt to decrement the integer. If the integer is zero, the thread will block until it is positive, and then unblock and decrement the integer.
  - A thread can call `up()` (also known as **V**) to increment the integer, which will always succeed.

  Unlike locks, semaphores have no concept of "ownership", and any thread can call `down()` or `up()` on any semaphore at any time.

2 Processes

2.1 Forks

How many new processes are created in the below program assuming calls to fork succeeds?

```
int main(void)
{
  for (int i = 0; i < 3; i++)
      pid_t pid = fork();
}
```

2.2 Process Stack Allocation

What can C print?

```
int main(void)
{
    int stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", stuff);
    if (pid == 0)
        stuff = 6;
}
```

2.3 Process Heap Allocation

What can C print?

```
int main(void)
{
    int* stuff = malloc(sizeof(int)*1);
    *stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", *stuff);
    if (pid == 0)
        *stuff = 6
}
```

2.4 Simple Wait

What can C print? Assume the child PID is 90210.

```
int main(void)
{
    pid_t pid = fork();
    int exit;
    if (pid != 0) {
        wait(&exit);
    }
    printf("Hello World\n: %d\n", pid);
}
```

2.5 Simple Pipe

Explain how the following code snippet uses `pipe()` to communicate between processes.

```
int main(void)
{
    char msg[16];

    int fildes[2];
    if (pipe(fildes) == -1)
            return -1;

    pid_t pid = fork();
    if (pid != 0) {
        strcpy(msg, "Go Bears!");
        write(fildes[1], msg, 16);
    } else {
        read(fildes[0], msg, 16);
        printf("%s\n", msg);
    }

    return 0;
}
```

3 Threads

3.1 Join

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    printf("MAIN\n");
    return 0;
}
```

How can we modify the code above to always print out "HELPER" followed by "MAIN"?

3.2 Thread Stack Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    int *num = (int*) arg;
    *num = 2;
    return NULL;
}

int main() {
    int i = 0;
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, &i);
    pthread_join(thread, NULL);
    printf("i is %d\n", i);
    return 0;
}
```

3.3 Thread Heap Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    return NULL;
}

int main() {
    char *message = malloc(100);
    strcpy(message, "I am the parent");
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, message);
    pthread_join(thread, NULL);
    printf("%s\n", message);
    return 0;
}
```

3.4 The Central Galactic Floopy Corporation

It's the year 3162. Floopies are the widely recognized galactic currency. Floopies are represented in digital form only, at the Central Galactic Floopy Corporation (CGFC).

You receive some inside intel from the CGFC that they have a Galaxynet server running on some old OS called x86 Ubuntu 14.04 LTS. Anyone can send requests to it. Upon receiving a request, the server forks a POSIX thread to handle the request. In particular, you are told that sending a transfer request will create a thread that will run the following function immediately, for speedy service.

```
void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient); // Thanks CS161

  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

Assume that there is some struct with a member `balance` that is `typedef`-ed as `account_t`.
Describe how a malicious user might exploit some unintended behavior.

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

3.5 Crowded Video Games

A recent popular game is having issues with its servers lagging heavily due to too many players being connected at a time. Below is the code that a player runs to play on a server:

```
void play_session(struct server s) {
    connect(s);
    play();
    disconnect(s);
}
```

After testing, it turns out that the servers can run without lagging for a max of up to 1000 players concurrently connected.

How can you add semaphores to the above code to enforce a strict limit of 1000 players connected at a time? Assume that a game server can create semaphores and share them amongst the player threads.

4 Threads and Processes

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *worker(void *arg) {
    int *data = (int *) arg;
    *data = *data + 1;
    printf("Data is %d\n", *data);
    return (void *) 42;
}

int data;
int main() {
    int status;
    data = 0;
    pthread_t thread;

    pid_t pid = fork();
    if (pid == 0) {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
    } else {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        wait(&status);
    }
    return 0;
}
```

How would you retrieve the return value of worker? (e.g. ”42”)

5 Pintos Lists

This section is intended to help you get more familiar with the pintos list abstraction, which will be used heavily in all three projects, as well as in homework 1. Consider the following code, which finds the sum of a traditional linked-list:

```
struct ll_node
{
    int value;
    struct ll_node *next;
}

/* Returns the sum of a linked list. */
int ll_sum(ll_node *start) {
    ll_node *iter;

    int total = 0;
    for (iter = start; iter != NULL; iter = iter->next)
        total += iter->value;

    return total;
}
```

Take a second to make sure you understand the structure of the for-loop, as this kind of iteration is key when dealing with linked lists.

Write code below that emulates the above code, but for pintos-style lists. That is, write a function that finds the sum of a pintos-style list. Some useful methods are listed below.

```
struct pl_node
{
    int value;
    struct list_elem elem;
}

/* Returns the sum of a pintos-style list of pl_nodes. */
int pl_sum(struct list *lst) {
    struct list_elem *iter;
    struct pl_node *temp;

    int total = 0;
```

```
    return total;
}
```

Here are some useful helper functions for pintos lists:

```
/* Given a struct list, returns a reference to the
 * first list_elem in the list. */
struct list_elem *list_begin(struct list *lst);

/* Given a struct list, returns a reference to the
 * last list_elem in the list. */
struct list_elem *list_end(struct list *lst);

/* Given a list_elem, finds the next list_elem in the list. */
struct list_elem *list_next(struct list_elem *elem);

/* Converts pointer to list element LIST_ELEM into a pointer to the
 * structure that LIST_ELEM is embedded inside. You must also
 * provide the name of the outer structure STRUCT and the member
 * name MEMBER of the list element. */
STRUCT *list_entry(LIST_ELEM, STRUCT, MEMBER);
```

Note that because `list_entry()` is actually defined as a preprocessor macro, it doesn't follow the normal rules of C functions, and introduces some interesting polymorphism.

If you need more help with the pintos list abstraction, check out the documentation in the pintos source code at `lib/kernel/list.h`. The documentation is very comprehensive, and you should refer to it as you do more with pintos lists.