jonmurata.me/attendance

Section 7: Wait + Exit in PintOS, Calling Conventions, Midterm Review

CS162

July 17, 2019

Contents

1 Wait and Exit

This problem is designed to help you with implementing wait and exit in your project. Recall that wait suspends execution of the parent process until the child process specified by the parameter id exits, upon which it returns the exit code of the child process. In Pintos, there is a 1:1 mapping between processes and threads.

1.1 Thinking about what you need to do

"wait" requires communication between a process and its children, usually implemented through shared data. The shared data might be added to struct thread, but many solutions separate it into a separate structure. At least the following must be shared between a parent and each of its children:

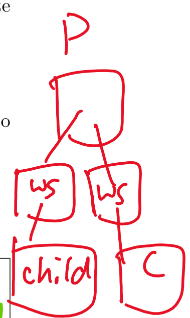
- Child's exit status, so that "wait" can return it.
- Child's thread id, for "wait" to compare against its argument.
- A way for the parent to block until the child dies (usually a semaphore).
- A way for the parent and child to tell whether the other is already dead, in a race-free fashion (to ensure that their shared data can be freed).

1.2 Code

Data structures to add to thread.h for waiting logic:

struct wait_status {
dead semaphore
list_elem
exit code
ref_count = 2
lock for ref_count
Child tid
}

thread
list of childrens
parent ws



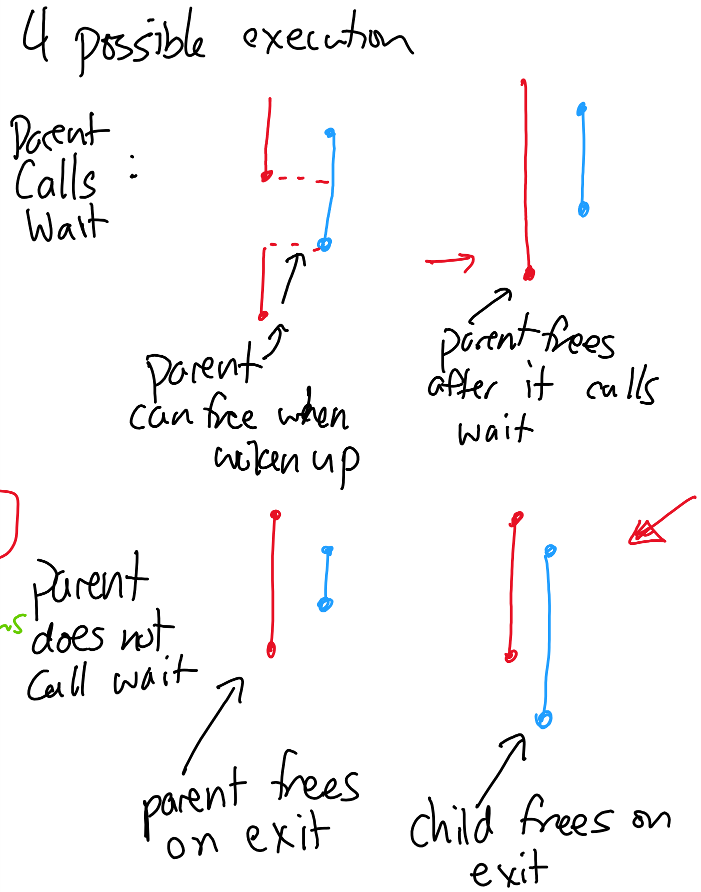
Implement wait:

- check list for ws, == child tid
- down semaphore
- get exit code
- free struct
- return exit code

Implement exit:

- put exit code in parent ws
- check ref_count
if 0 refs ⇒ free
else
sema_up

for all ws in children
decrement ref_count
If 0 refs ⇒ free



2 Calling Conventions and Argument Passing

2.1 Calling Conventions

Sketch the stack frame of `helper` before it returns.

```
void helper(char* str, int len) {
 char word[len];
 strncpy(word, str, len);
 printf("%s", word);
 return;
}

int main(int argc, char *argv[]) {
 char* str = "Hello World!";
 helper(str, 13);
}
```

3 Midterm Review

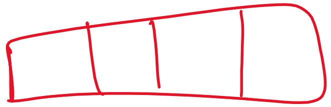
3.1 Signals and Forks

Given the following code, write out all possible outputs.

```
pid_t pid; int counter = 3;

void rem(int signum) {
    counter *= 5;
    printf("counter: %d\n", counter);
    kill(pid, SIGUSR1);
}

void emilia(int signum) {
    counter += 5;
    printf("counter: %d\n", counter);
    exit(0);
}

int main() {
    pid_t p; int status;
    signal(SIGUSR1, rem);
    if ((pid = fork()) == 0) {
       signal(SIGUSR1, emilia);
       kill(getppid(), SIGUSR1);
       while(1);
    }
    counter = 1000;
    printf("counter: %d\n", counter);
}
```

3.2 Spring 2017, MT1 P5

Next Saturday is the international day of Poker. As the owner of the largest poker website worldwide you expect a large number of games being played (and finishing) at any point in time in your website. Consider that players can play more than one game at a time and any two players can play against each other in more than one game simultaneously. For simplicity, we consider each game has exactly two players. The backend system of your poker website contains the following multi-threaded code:



```
queue games_finished_queue;
lock_t games_finished_lock;
semaphore games_to_process_sem;

typedef struct Game {
    ....
} Game;

typedef struct Player {
    lock_t lock;
    uint64_t n_chips;
    uint64_t unique_id;
} Player;

void finish_game(Game *game) {
    lock_acquire(&games_finished_lock);
    enqueue(&games_finished_queue, game);
    lock_release(&games_finished_lock);
    sema_up(&games_to_process_sem);
}

void process_finished_games() {
    lock_acquire(&games_finished_lock);
    sema_down(&games_to_process_sem);
    Game *g = pop_queue_front(&games_finished_queue);
    move_chips(g->player1, g->player2, g->n_chips);
    lock_release(&games_finished_lock);
}

void move_chips(Player *player1, Player *player2, uint64_t n_chips) {
    lock_acquire(&player1->lock);
    lock_acquire(&player2->lock);
    player1->n_chips -= n_chips;
    player2->n_chips += n_chips;
    lock_release(&player2->lock);
    lock_release(&player1->lock);
}
```

(a) Identify two places in the code where deadlock can occur. If deadlock occurs, use no more than two sentences to explain why it occurs.

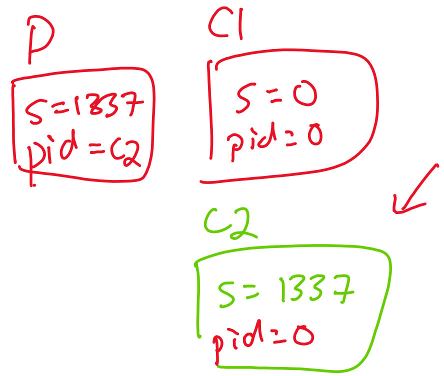
(b) Use the space bellow to change `process_finished_games()` and `move_chips()` (or copy if correct) to ensure no deadlocks can occur. Explain succinctly why no deadlock can occur with the newly modified code. Note: a single lock at the beginning and end of `move_chips` is not an accepted solution.

```
void process_finished_games() {
    ___________________________________________;
    ___________________________________________;
    Game* g = pop_queue_front(&games_finished_queue);
    move_chips(g->player1, g->player2, g->n_chips);
    ___________________________________________;
}

void move_chips(Player* player1, Player* player2, uint64_t n_chips) {
    ___________________________________________;
    ___________________________________________;
    ___________________________________________;
    ___________________________________________;
    ___________________________________________;
    ___________________________________________;
    ___________________________________________;
    player1->n_chips -= n_chips;
    player2->n_chips += n_chips;
    ___________________________________________;
    ___________________________________________;
}
```

3.3 Fall 2017, MT1 P2

Consider the following C program. Assume that all system calls succeed when possible.

```
void *rem(void *args) {
    printf("Blue: %d\n", *((int *) args));
    exit(0);
}

void *ram(void *args) {
    printf("Pink: %d\n", ((int *) args)[0]);
    return NULL;
}

int main(void) {
    pid_t pid; pthread_t pthread; int status; //declaring vars
    int fd = open("emilia.txt", O_CREAT|O_TRUNC|O_WRONLY, 0666);
    int *subaru = (int *) calloc(1, sizeof(int));
    printf("Original: %d\n", *subaru);

    if (pid = fork()) {
        *subaru = 1337;
        pid = fork();
    }

    if (!pid) {
        pthread_create(&pthread, NULL, ram, (void*) subaru);
    } else {
        for (int i = 0; i < 2; i++)
            waitpid(-1, &status, 0);
        pthread_create(&pthread, NULL, rem, (void*) subaru);
    }
    pthread_join(pthread, NULL);

    if (*subaru == 1337)
        dup2(fd, fileno(stdout));
    printf("All done!\n");
    return 0;
}
```



(a) Including the original process, how many processes are created? Including the original thread, how many threads are created?

3P, 6T

(b) Provide all possible outputs in standard output.

Original 0
Pink: 1337
Pink: 0
All done!
Blue 1337

Orang 0
Pink: 0
Pink: 1337
All done!
Blue 1337

Orng 0
0
All done
Pink 1337
Blue 1337

(c) Provide all possible contents of `emilia.txt`.

All done!

(d) Suppose we deleted line 28 (`if *subaru == 1337`), how would the contents of `emilia.txt` change (if they do)?

(e) What if, in addition to doing the change in part (d), we also move line 12 (where we open the file descriptor) between lines 19 and 20 (exactly after the first if statement)? What would `emilia.txt` look like then?