Andrew Ma

Lab 4

CPE 435

2/8/21

Theory

This lab is for exploring IPC (inter-process communication) using shared memory. A process can create a shared memory segment and attach it to a local pointer, and other processes can attach to that shared memory segment with their local pointers and read and update data in the shared data structure. Usually there is a flag in the shared data structure so a process can set it to a different value to take or release control and other processes can wait for a certain flag value to do their designated operations. Shared memory is more efficient than pipes because there is no intermediate copying of data like in pipes (where we wrote and read data using file descriptors). Shared memory allows for random access since it is just a block of allocated memory, unlike a sequential byte stream in pipes. Shared memory also allows for a many-to-many mechanism, where many processes can attach to the same segment, and there can be multiple segments created at the same time. Pipes provide a one-to-one mechanism. There is no synchronization in shared memory so we set a flag variable, but pipes provide synchronization. Shared memory segments can have limits set by a sysadmin like maximum and minimum size in bytes, or maximum number of shared memory segments that can be created per system or process.

The functions and types for creating and using shared memory segments are in sys/ipc.h and sys/shm.h.

The int shmget(key_t key, size_t size, int shmflg) function is used to allocate a shared memory segment. It returns the identifier of the shared memory segment associated with the key argument. It can create a new shared memory segment by passing in a flag with IPC_CREAT, and it can be used to get an existing shared memory segment by passing in a flag of 0. The second argument is for the size of the shared memory segment rounded up to a multiple of PAGE_SIZE.

The void* shmat(int shmid, const void* shmaddr, int shmflg) function is used to attach to a shared memory segment associated with the id returned by the shmget() function, and it returns the pointer in the calling process address space. If shmaddr is NULL, the segment is attached at the first available address that the system chooses. If shmaddr is not NULL and shmflg is 0, then the segment is attached at shmaddr. We usually can let the system choose what address in the calling process to use, so we can pass in NULL for shmaddr. We also want to cast

the returned void ptr to a ptr of our shared struct in order to access the member variables by name.

The int shmctl(int shmid, int cmd, struct shmid_ds* buf) function is used to perform the cmd control operation in the shared memory segment identified by shmid. The buf arg is a ptr to a shmid_ds structure that can be used to read properties and permissions of the shared memory segment using the cmd IPC_STAT or update the properties and permissions using the cmd IPC_SET. The only cmd we use in the code is IPC_RMID, and it marks the shared memory segment to be destroyed. The segment will be destroyed only after the last process detaches it (with shmdt()). If IPC_RMID is not used, it can cause memory leaks because the shared memory segments will not be destroyed on normal exit.

The int shmdt(const void* shmaddr) function is used to detach the local ptr in the calling process from the shared memory segment that was attached with shmat(). It will decrement the shm_nattach number in the shared memory data structure. This is important because shmctl() with IPC_RMID will only destroy the shared memory segment if the shm_nattach is 0, or no calling processes have pointers that are still attached to the shared memory segment.

Observations

ProcessA can set value1, value2, and flag (by selecting y for terminate it sets flag to -1, causing both processes to terminate)

ProcessB sets sum to the sum of value1 and value2, and it displays the result.

Before terminate

After terminate

If process A not running (no shared memory segment created), then process B errors since it only tries to get an existing shared memory segment.

Conclusion

The program worked as expected. Process A could set the two values and as soon as the process A sets the flag to 1, process B immediately gets the two values in the shared memory, adds them, sets the sum in the shared memory, and prints the sum. Process A is the one creating a new shared memory segment each time, and process B is trying to attach to an existing shared memory segment. This works fine and the error condition for process B in case there is no existing shared memory segment is shown in the screenshot above. Also, the shmctl(IPC_RMID)

and shmdt() are used before exit to ensure that allocated shared memory segments are destroyed and don't leak memory. This is verified using the ipcs command on linux. From this lab, I learned how processes can create shared memory segments and communicate with each other using a flag in the shared memory for synchronization. I also learned about the various benefits of shared memory over pipes, such as the possibility for many-to-many and the speed of random access.

Source Code

```c
// processA.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <errno.h>
#include <limits.h>
#include <signal.h>

#include "sharedInfo.h"

#define userInputBufferLength 20

void INThandler(int);
int getUserInputInt(char* const buff, int const buff_len, int* const
intVar);
struct info* ctrl;
int id;


int main(void) {
    signal(SIGINT, INThandler);
```

```
    // create shared memory segment using shmget()
    // id: file identifier of shared memory segment
    // if key is shared, any process can attach to this shared memory
segment
    // 0666 is file permission for shared memory segment
    id = shmget(key, MSIZ, IPC_CREAT | 0666);
    if (id < 0) {
        // error
        fprintf(stderr, "Error with shmget() creating shared memory
segment\n");
        exit(1);
    }

    // attach local ptr to shared memory segment created using shmat()
    ctrl = (struct info*)shmat(id, 0, 0);
    if (ctrl <= (struct info*)(0)) {
        // error
        fprintf(stderr, "Error with shmat() attaching to shared memory
segment\n");
        // mark the segment to be destroyed
        // segment will actually be destroyed only after last process
detaches from it
        shmctl(id, IPC_RMID, NULL);
        exit(1);
    }

    // initial values
    ctrl->value1 = 0;
    ctrl->value2 = 0;
    ctrl->sum = 0;
    ctrl->flag = 0;

    char userInputValue1[userInputBufferLength],
userInputValue2[userInputBufferLength],
userInputTerminate[userInputBufferLength];
```

```
    while (1) {
        // get user input and save to value1, and value2
        printf("\nTerminate? [y/N] ");
        fflush(stdout);

        fgets(userInputTerminate, userInputBufferLength, stdin);
        //// remove \n at end (or can just do it in comparison)
        //userInputTerminate[strlen(userInputTerminate) - 1] = '\0';

        if (strcmp(userInputTerminate, "y\n") == 0) {
            // set ctrl->flag to -1 to tell processB to also terminate
            ctrl->flag = -1;

            break;
        }



        printf("Enter value1: ");
        fflush(stdout);

        int value1 = 0;
        if (getUserInputInt(userInputValue1, userInputBufferLength,
&value1) == -1) {
            continue;
        }
        else {
            ctrl->value1 = value1;
        }



        printf("Enter value2: ");
        fflush(stdout);

        int value2;
        if (getUserInputInt(userInputValue2, userInputBufferLength,
&value2) == -1) {
```

```
            continue;
        }
        else {
            ctrl->value2 = value2;
        }

        // when processA done, sets flag to 1, telling processB it can
proceed
        ctrl->flag = 1;
    }

    // When finished
    // shmdt(): detaches shared memory segment at ctrl  from the address
space of the calling process
    shmdt(ctrl);

    // mark the segment to be destroyed
    // segment will actually be destroyed only after last process
detaches from it
    shmctl(id, IPC_RMID, NULL);

    exit(0);
}


int getUserInputInt(char* const buff, int const buff_len, int* const
intVar) {
    /* Parameters:
        buff (ptr to user input buffer)
        buff_len (length of user input buffer)
        intVar (ptr to intVar where final result would be saved if user
entered good int,
            otherwise intVar won't be changed if bad int
        )

        returns -1 if user entered bad int
```

```
        returns 0 if user entered good int
     */


    char* end = NULL;

    fgets(buff, buff_len, stdin);
    // remove \n character than fgets keeps
    buff[strlen(buff) - 1] = '\0';

    errno = 0;
    const long longVar = strtol(buff, &end, 10);

    // if ((end == buff) || ('\0' != *end) || ((LONG_MIN == longVar ||
LONG_MAX == longVar) && ERANGE == errno) || (longVar > INT_MAX) ||
(longVar < INT_MIN)) {
    if (end == buff) {
        fprintf(stderr, "%s: not a decimal number\n", buff);
        return -1;
    }
    else if ('\0' != *end) {
        fprintf(stderr, "%s: extra characters at end of input: %s\n",
buff, end);
        return -1;
    }
    else if ((LONG_MIN == longVar || LONG_MAX == longVar) && ERANGE ==
errno) {
        fprintf(stderr, "%s out of range of type long\n", buff);
        return -1;
    }
    else if (longVar > INT_MAX) {
        fprintf(stderr, "%ld greater than INT_MAX\n", longVar);
        return -1;
    }
    else if (longVar < INT_MIN) {
        fprintf(stderr, "%ld less than INT_MIN\n", longVar);
```

```
        return -1;
    }
    else {
        // set the intVar to valid user input
        *intVar = (int)longVar;
        return 0;
    }
}

void INThandler(int sig) {
    // shmdt(): detaches shared memory segment at ctrl  from the address space of the calling process
    shmdt(ctrl);

    // mark the segment to be destroyed
    // segment will actually be destroyed only after last process detaches from it
    shmctl(id, IPC_RMID, NULL);

    exit(0);
}
// can use `ipcs` to monitor created shared memory segments
```

```
// processB.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "sharedInfo.h"


int main(void) {
    // get id of already created shared memory segment
```

```
    int const id = shmget(key, MSIZ, 0);
    if (id < 0) {
        // error
        fprintf(stderr, "Error with shmget() getting shared memory
segment\n");
        exit(1);
    }

    //attach local ptr to shared memory
    struct info* ctrl = (struct info*)shmat(id, 0, 0);
    if (ctrl <= (struct info*)(0)) {
        // error
        fprintf(stderr, "Error with shmat() attaching to shared memory
segment\n");
        exit(1);
    }

    while (1) {
        // wait for processA to set flag to 1 for done changing value1
and value2
        while (ctrl->flag != 1 && ctrl->flag != -1);

        if (ctrl->flag == -1) {
            // if processA set flag to -1, then we want to exit after
cleanup
            break;
        }

        ctrl->sum = ctrl->value1 + ctrl->value2;
        // display result
        printf("Sum: %d\n", ctrl->sum);
        fflush(stdout);

        // setting flag to 0
        ctrl->flag = 0;
    }
```

```
    // detach ptr from shared memory
    shmdt(ctrl);
    exit(0);
}
```

```
// sharedInfo.h
#ifndef INFO_H
#define INFO_H

#include <sys/types.h>

struct info {
    char flag; // set by pA (if -1, then both processes terminate)
    int value1, value2; // set by pA
    int sum; // set by pB, and displayed by pB
};

key_t key = 12345;
size_t MSIZ = sizeof(struct info);

#endif
```