CS 24000 - Programming In C

Week 14: Use GDB to debug multiple processes;

Shared memory for interprocess communication (IPC)

Zhiyuan Li
Department of Computer Science
Purdue University, USA
• Unfortunately, gdb does have a special support for debugging child processes.
• After fork(), gdb continues executing the parent process and cannot monitor the child process.
• In order to debug a child process, we need to start another gdb run and “attach” the child process ID.
• Suppose the child process is running
  – gdb <program_name>
  – attach child_pid  // find out by command ps
• However, we need to insert a statement for the child process to wait
  – Otherwise before gdb could attach child_pid, the child process may have run past the code segment of interest, or may even have exited.
  – So we need to suspend the execution in the child process when in debugging mode
    • Use a sleep() call, or
    • Use an infinite loop
  – When we compile using the “-g” switch, we also define some debugging environment variable
• The gdb environment might not respond to gdb commands well (e.g. step, well)
• It may also not respond to interrupt key strokes well, e.g. ctl_C, ctl_Z
• In such a situation, we will need to issue the “kill -<SIGNAL> <PID>” command from another login window
• We can find <PID> by running “ps –U <username>”
int main(void)
{
    pid_t child_PID;
    int i=0;

    child_PID = fork(); // assume fork succeeds

    We can
(i) Start gdb to run parent process
    (i) #break main \set a break point in main()
    (ii) #step
(ii) Start gdb to run child process, which will stop inside the infinite loop
(iii) #set variable i=0   \to get out of the infinite loop

    child_PID == 0 { // child process
        printf("In Child Process, infinite loop\n");
        \ifdef DEBUG
            i = 1;
            while (i) {i=1;}
        \endif
    } else { //Parent process
        printf("In Parent Process after fork%d\n", child_PID);
        exit(0);
    }
}
• **SIGINT**
  – This is the signal that has the effect of entering the key of **CTRL-C**.
  – Try issue a command “sleep 30 &”
  – Find the process id
  – Kill the background job by “kill –INT <pid>”

• **SIGSTOP**
  – Has the effect of entering the key of **CTRL-Z**.
  – Try issue a command “sleep 30 &”
  – Find the process id
  – Stop (without terminating) the background job by “kill –STOP <pid>”
• Next we’ll run some debugging sessions.
  
  – First, we run a program (forkflush.c) in which we forgot to flush the file buffer in the writer process and therefore the (child) reader process fails to read the content
  
  – Next, we run another program (forksharefile.c) in which the parent process tries to write to a file that is to be read by the child process (simultaneously) as a way to communicate data, but it does not work well.
  
  • Which motivates the use of pipe.
Shared memory for data communication between processes

• The use of pipe() for data communication is quite constrained.

• Therefore are two main ways to communicate data between processes that are more general
  — Message passing (using message queues)
  — Shared memory

• Each has pros and cons and the debate has continued for decades

• We will discuss shared memory this semester
  — This follows the standard syllabus for CS240
Creating Shared Memory

```c
int shmget(key_t key, size_t size, int shmflg);
```

- Among the processes that shared the same shared memory segment, at least one of them must “create” that segment
  - Specifying `shmflg` to be “IPC_CREAT | 0666”
    - “mode_flags (least significant 9 bits) specifying the permissions granted to the owner, group, and world.
    - Specify “0666” to ensure access permission
  - All these processes must use the same key to call the `shmget` function
  - The creator sets the size (in bytes)
    - The system will round it up to a multiple of the page size
  - The other callers can specify the size no greater than the one used during creation
What should be the key?

• The system is quite permissive with the key values
  – You can use any integer as the key
  – Obviously this creates a security concern
• For related processes, e.g. parent/child processes, one can use the constant IPC_PRIVATE
• For unrelated processes, it is up to the programmer(s) to set up the key in some discreet fashion
  – E.g. use the ftok() system call to generate a key based on some unpublicized file path name
  – For simplicity, we will just use arbitrary integers in our examples
The shmget() call (cont’d)

• The call returns an integer, to id the segment
• shmflg is a rights mask (0666) OR’d with one of the following:
  – IPC_CREAT will create or attach
  – IPC_EXCL creates new or it will report error if it exists
Some predefined constants

- **SHMALL**
  - System wide maximum of shared memory pages
- **SHMMMAX**
  - Maximum size in bytes for a shared memory segment
- **SHMMIN**
  - Minimum size in bytes for a shared memory segment
- **SHMMNMI**
  - System wide maximum number of shared memory segments
Attach shared memory segment to a pointer

• Just like when we use malloc()
  – So we can step through the shared memory via the pointer
• `void *shmat(int shmid, const void *shmaddr, int shmflg);`
  – On success `shmat()` returns the address of the attached shared memory segment;
  – on error (void *) -1 is returned, and `errno` is set to indicate the cause of the error.
  – If `shmaddr` is NULL, the system chooses a suitable (unused) address at which to attach the segment.

• Example:
  • if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
  •     perror("error when call shmat");
  •     exit(1);
  • }
An example with two unrelated processes

• For simplicity, we begin with an example of using shared memory for data communication between two unrelated processes
  – Two separately issued command
    • A server
    • A client

• (This example is adapted from an example from the internet)
#include <unistd.h>    // server.c
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

#define SHMSIZE 1024

int main()
{
    int i;
    int shmid;
    key_t key;
    int *shm, *s;

    key = 12345678;
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("error when call shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (int *) -1)
        perror("error when call shmat");
    exit(1);
    s = shm;
    for (i = 1; i <= 50; i++)
        *s++ = i;
    *s = 0;
    exit(0);
}
#include .....    // client.c
#define SHMSIZE 1024

main()
{
    int shmid;
    key_t key;
    int *shm, *s;
    key = 12345678;
    /** Find the segment. What if client runs first?
     */
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("call shmget");
        exit(1);
    }
    if ((shm = shmat(shmid, NULL, 0)) == (int *)-1) {
        perror("call shmat");
        exit(1);
    }
    for (s = shm; *s != 0; s++)
        printf("%d
", *s);
    putchar('\n');
    exit(0);
}
• From the example, we see that shmget() and shmat() together allocates a (shared) memory block to store an array of something (int, char, struct, etc)
  – Compare with (void *) -1, with void replaced by int, char, etc
  – It is up to the programmer to allocate sufficient memory size
• We run server first (in the background)
• Then we run client
• We see client prints out the list of integers before terminates
• By issuing a command “ipcs”, we can see that the shared memory block still exists after both programs terminate
  – If you run the client program again, it will still be able to read and print the data (the data also exist)
• This is not good, we must remove the block
  – We can do this by a Unix command “
    – Using system call **ipcrm** *(by specifying the key or id of the block)*
    – **Better yet, we remove the block within the program either by server or by client. If there are multiple clients, better by the server.**
    – **For simplicity, we add to client “shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0);”**
• Then the server terminates
• The remaining issue:
  – The shared memory stays in the system
  – To remove it now, we need to issue a command “ipcrm”.
  – To identify the shared memory to remove by ipcrm, there are several options (read the man page), e.g. “–M key”
• It is a better practice for remove the shared memory before program terminates
  – Uncomment the “shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0); “ call and rerun the programs
Shared Memory Control

```c
struct shmid_ds {
    int shm_segsz; /* size of segment in bytes */
    __time_t shm_atime; /* time of last shmat command */
    __time_t shm_dtime; /* time of last shmdt command */
    ...
    unsigned short int __shm_npages; /* size of segment in pages */
    msgqnum_t shm_nattach; /* number of current attaches */
    ...
} /* pids of creator and last shmop */
};
```

- **int shmid_ds (int shmid, int cmd, struct shmid_ds * buf);**
- cmd can be one of:
  - IPC_RMID destroy the memory specified by shmid
  - IPC_SET set the uid, gid, and mode of the shared mem
  - IPC_STAT get the current **shmid_ds struct** for the queue
Flags in shmat() call

• Usually 0
• Other possibilities
  – SHM_RDONLY sets the segment as read-only
  – SHM_RND sets page boundary access
  – SHM_SHARE_MMU set first available aligned address
Related processes sharing memory

• This is more complex in some sense
  – Because we need more synchronization effort
    • Recall that in the client/server example, we artificially let server run first
  – If the parent process plays the server role
    • The main process, after writing to shared memory, must give go ahead to the child process to read
  – We can create another shared memory block to do such “hand-shaking”

• The simpler parts with related process are
  – We can use **IPC_PRIVATE** to get a unique anonymous key
  – Let parent process create and attach the shared memory blocks
  – Child processes will inherit
• We draw the time line and explain the parallel events in both processes
• The following program runs a parent process (server) and a child process (client)
• Two shared memory segments are created
  – One used for synchronization
    • Not the most efficient way, but simple and intuitive
    • We will discuss semaphores for synchronization later
```c
#include ...
#define ...
int main()
{
    int i;
    int shmid1, shmid2;
    key_t key;
    int *shm1, *shm2, *s;
    pid_t child_PID;

    if ((shmid1 = shmat(IPC_PRIVATE, SHMSIZE, 0666)) < 0) {
        perror("shmat");
        exit(1);
    }
    if ((shm1 = shmat(shmid1, NULL, 0)) == (int *)-1) {
        perror("shmat");
        exit(1);
    }
    *shm1 = 0;

    if ((shmid2 = shmat(IPC_PRIVATE, SHMSIZE, 0666)) < 0) {
        perror("shmat");
        exit(1);
    }
    if ((shm2 = shmat(shmid2, NULL, 0)) == (int *)-1) {
        perror("shmat");
        exit(1);
    }
    child_PID = fork();
    if (child_PID < 0) { // error
```
printf("\n Fork failed\n");
    exit (1);

}

if(child_PID == 0) { // child process
    /* wait for parent to give a go ahead */

    while (*shm1 != 1)
        sleep(1);

    for (s = shm2; *s != 0; s++)
        printf("%d\n", *s);
    putchar('\n');

    *shm1 = 0;
    exit(0);
} // end child process

else { // parent process

    s = shm2;

    for (i = 1; i <= 50; i++)
        *s++ = i;
    *s = 0;

    *shm1 = 1; // Give child process go-ahead to read

    /*
        * We wait for the child process give a go
        ahead terminate
        */
    while (*shm1 != 0)
        sleep(1);
    exit(0);
} // end of parent process
}
• Run this program
• We see parent process successfully passes data to the child process
• Remaining issues
  – Again, there will be shared memory blocks staying in the system
    • With 0000000 key because of the anonymity
    • We can remove them by ipcrm –m id
  – Again, it is better practice to remove them before terminating
Quiz 10 #1

• Which statement is true after a program successfully executes “pipe (mypipe)”?
• (1) a pipe named “mypipe” is created
• (2) an unnamed pipe is created
• (3) a pointer variable “mypipe” will point to a pipe that has been created
• (4) the status of this call is written to the variable mypipe
• Answer (b)
• Which statement is the most accurate after a program successfully executes “pipe (mypipe)”? (If more than one statement is true then you should choose the answer that states so)
  • (1) an array, mypipe[2], will store an integer file ID in each of its two elements
  • (2) mypipe[1] is the read end of the pipe created and mypipe[0] is the write end
  • (3) mypipe[1] is the write end of the pipe created and mypipe[0] is the read end
  • (4) both (1) and (2) are correct
  • (5) both (1) and (3) are correct
  • (6) none of the above is correct
• Answer (5)
Which statement is correct after a program executes “pipe (mypipe)”?

(a) A returned value of 0 means the call failed
(b) A return value of -1 means the call failed
(c) This is implementation dependent
• Answer (b)