AUP : Assignment - 7 [Signals]

Aditi Rajendra Medhane 111803177

18th October 2021

Q1

Create a child process. Let the parent sleeps of 5 seconds and exits. Can the child send SIGINT to its parent if exists and kill it? Verify with a sample program.

Code

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

int main(void) {
        int child_pid;

        if ((child_pid = fork()) == -1) {
                //fork failed
                perror("fork");
                return errno;
        }
        else if (child_pid) {
                //parent
                sleep(5);
                printf("Parent is Alive\n");
        }
        else {
                //child
                if (kill(getppid(), SIGINT) == -1) {
                        perror("SIGINT to parent");
                        return errno;
                }
                printf("Child sent kill signal\n");
                sleep(5);
                printf("Child slept for time equal to parent\n");
        }

        return 0;
}
```

Explanation

Parent is alive not printed, parent killed by SIGINT sent by child

Output

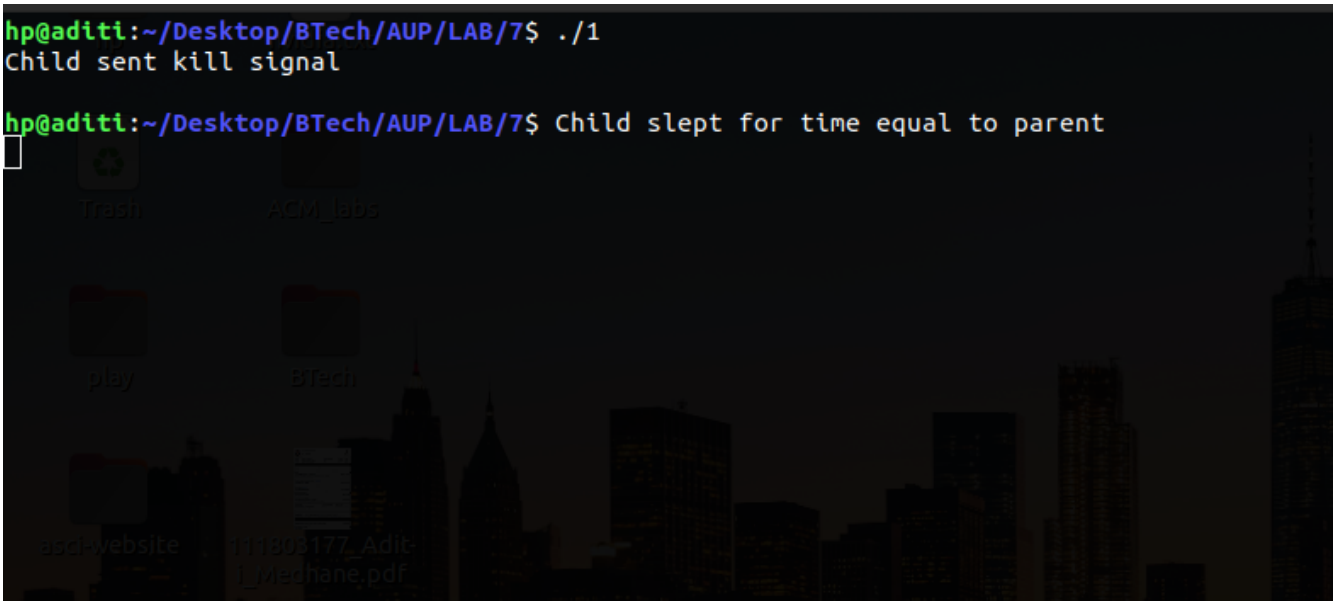


Figure 1: Output 1

Q2

Create a signal disposition to catch SIGCHLD and in the handler function display some message. Create a child process and let the child sleeps for some time and exits. The parent calls a wait() for the child. Display the return value of wait() to check success or failure. If failure, display the error number. Run the program:

a. Normal way executing in the foreground

b. Run as a back ground process and send SIGCHLD to it from the shell

Code

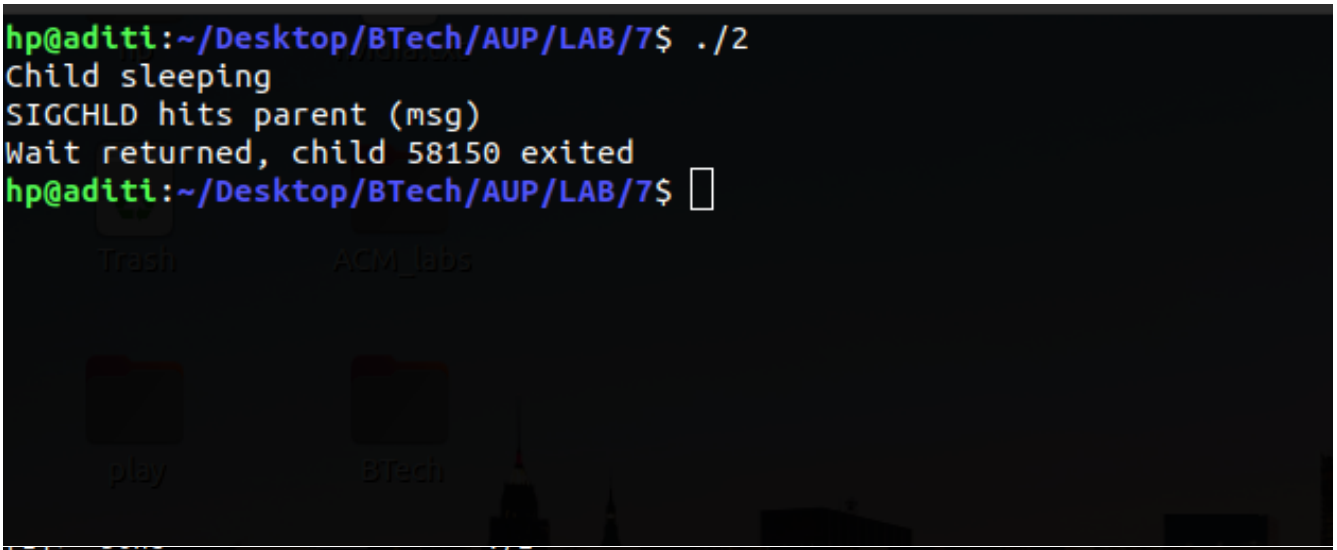
```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <stdio.h>

void print_sigchld_msg(int sig_number) {
        printf("SIGCHLD hits parent (msg)\n");
}

int main(void) {
        pid_t child_id;

        int dead_child;
        int status;

        /* set handler for SIGCHLD */
        signal(SIGCHLD, print_sigchld_msg);

        if ((child_id = fork()) == -1) {
                perror("fork");
                return errno;
        }
        else if (child_id) {
                /* parent */

                if ((dead_child = wait(&status)) == -1) {
                        perror("wait failed");
```

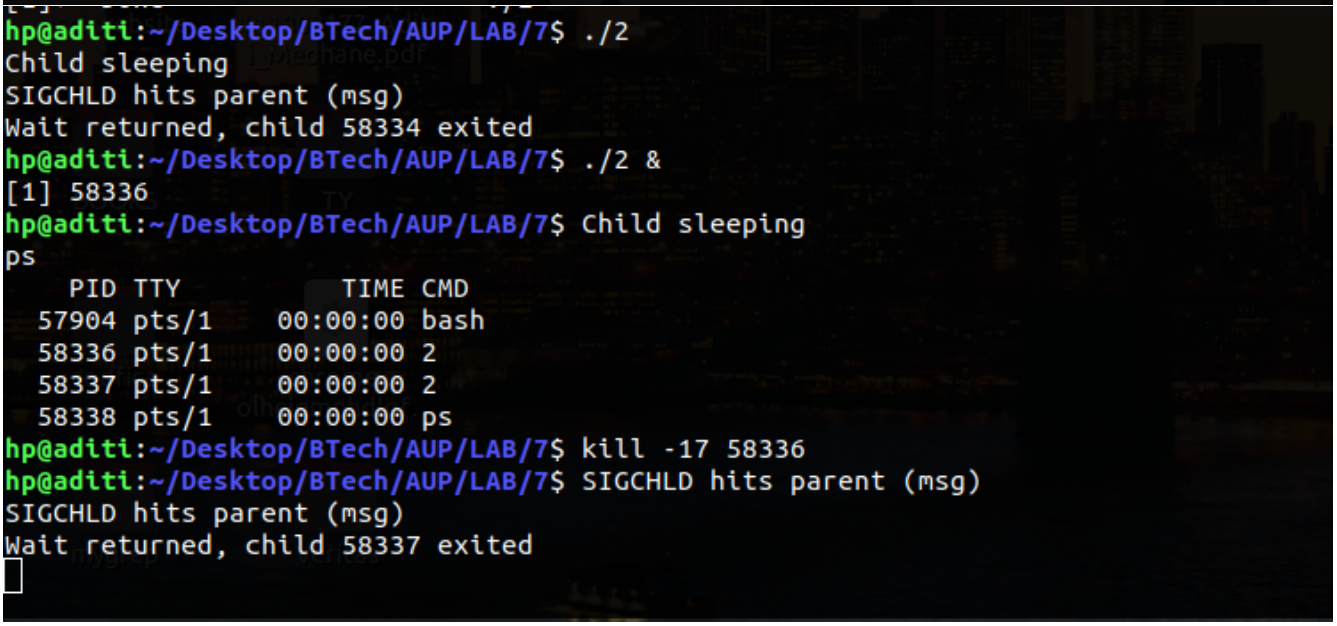
```
30                      return errno;
31              }
32              else {
33                      printf("Wait returned, child %d exited\n", dead_child);
34              }
35      }
36      else {
37              /* Child */
38              printf("Child sleeping\n");
39              sleep(10);
40      }
41
42      return 0;
43  }
```

Explanation

- Image 2_1 : Normal way of executing the program, SIGCHLD hits once
- Image 2_2 : Run as a back ground process and send SIGCHLD to it from the shell, SIGCHLD hits twice

Output





Q3

You have to create a process tree as shown below. Then you let the parent process create a process group of (3, 4, 5) so that it sends a signal to this group. Print appropriate messages.

Code

```c
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define SHARED_ARR "shared_array"
#define SEMAPHORE_FORKING "fork_semaphore"
#define N 6

/* to ensure that fp can be used by all processes */
static int fp;

void print_message(int signo) {
        printf("%d got hit by signal %d\n", getpid(), signo);
}

void examine_child(int pid) {
    int status;

    if (waitpid(pid, &status, 0) == -1) {
            perror("wait");
            exit(errno);
    }

    if (WIFEXITED(status)) {
            printf("%d exited with %d\n", pid, WEXITSTATUS(status));
    }
    else if (WIFSIGNALED(status)) {
            printf("%d killed by signal %d\n", pid, WTERMSIG(status));
    }
    else {
            printf("%d died in some other way\n", pid);
    }
}

void *get_shared_arr() {
        void *buf;
        /* asssert: fp is a file descriptor which points to the shared memory,
         * and is inherited by all processes */
        if ((buf = mmap(NULL, sizeof(int) * N, PROT_READ | PROT_WRITE, MAP_SHARED, fp, 0)) == MAP_FAILED) {
                perror("shared memory mmap() failed");
                exit(errno);
        }
        return buf;
}


int main(void) {

        // PROCESS 0
        // can see the pids of {1, 2}
        sem_t *fork_sem, *pgid_sem, *exit_sem;
        int *child_pid;
```

```
61          int ret;
62
63          if (signal(SIGUSR1, print_message) == SIG_ERR) {
64                  perror("SIGUSR1");
65                  return errno;
66          }
67
68          // link shared memory address to process- this will be visible in children
69          if ((child_pid= mmap(NULL, sizeof(int) * N,
70                                          PROT_READ | PROT_WRITE,
71                                          MAP_SHARED | MAP_ANONYMOUS,
72                                          -1, 0)) == MAP_FAILED) {
73
74                  perror("mmap");
75                  return errno;
76          }
77
78          // create a shared memory map for semaphore
79          if ((fork_sem = mmap(NULL, sizeof(sem_t),
80                                          PROT_READ | PROT_WRITE,
81                                          MAP_SHARED | MAP_ANONYMOUS,
82                                          -1, 0)) == MAP_FAILED) {
83
84                  perror("semaphore");
85                  return errno;
86          }
87          /* initialize semaphore with initial value 0- when 5 and 3 get
88           * created, this is incremented by 1 each. 0 will synchronize by calling down() on this twice */
89          if (sem_init(fork_sem, 1, 0) == -1) {
90                  perror("semaphore initialization");
91                  return errno;
92          }
93
94          // create a shared memory map for semaphore
95          if ((exit_sem = mmap(NULL, sizeof(sem_t),
96                                          PROT_READ | PROT_WRITE,
97                                          MAP_SHARED | MAP_ANONYMOUS,
98                                          -1, 0)) == MAP_FAILED) {
99
100                 perror("semaphore");
101                 return errno;
102         }
103
104         /* initialize semaphore with initial value 0- when signals have been sent,
105          * this will be incremented 5 times, wherin all processes will exit */
106         if (sem_init(exit_sem, 1, 0) == -1) {
107                 perror("semaphore initialization");
108                 return errno;
109         }
110
111         // create a shared memory map for semaphore
112         if ((pgid_sem = mmap(NULL, sizeof(sem_t),
113                                         PROT_READ | PROT_WRITE,
114                                         MAP_SHARED | MAP_ANONYMOUS,
115                                         -1, 0)) == MAP_FAILED) {
116
117                 perror("semaphore");
118                 return errno;
119         }
120
121         /* initialize semaphore with initial value 0- when all processes have
122          * called setpgid, the parent can send signals */
123         if (sem_init(pgid_sem, 1, 0) == -1) {
124                 perror("semaphore initialization");
125                 return errno;
126         }
127
```

```
128         /* assert: now each child will have access to the semaphore, unless the
129          * memory region is purposely unliked */
130
131
132         if ((ret = fork()) == -1) {
133                 perror("fork 1");
134                 return errno;
135         }
136         else if (!ret) {
137                 // CHILD 1
138                 // can see pids of {5}
139                 // child_pid = (int *)get_shared_arr();
140
141                 if ((ret = fork()) == -1) {
142                         perror("fork 5");
143                         return errno;
144                 }
145                 else if (!ret) {
146                         // CHILD 5
147                         // can see pids of {}
148                         // child_pid = (int *)get_shared_arr();
149
150                         // wait for the PID of 3 to be available
151                         if (sem_wait(fork_sem) == -1) {
152                                 perror("P operation in 5");
153                         }
154
155                         // set own process group to 3
156                         if (setpgid(0, child_pid[3]) == -1) {
157                                 perror("setpgid(5, 3)");
158                                 return errno;
159                         }
160
161                         if (sem_post(pgid_sem) == -1) {
162                                 perror("setpgid(5, 3) done synchronization");
163                                 return errno;
164                         }
165
166
167                         // wait for 0 to allow exiting
168                         if (sem_wait(exit_sem) == -1) {
169                                 perror("V operation in 4");
170                                 return errno;
171                         }
172
173                         printf("%d is child of %d\n", getpid(), getppid());
174
175                         return 0;
176                 }
177                 child_pid[5] = ret;
178
179
180
181                 // wait for 0 to allow exiting
182                 if (sem_wait(exit_sem) == -1) {
183                         perror("V operation in 2");
184                         return errno;
185                 }
186
187                 printf("%d is child of %d\n", getpid(), getppid());
188
189                 examine_child(child_pid[5]);
190
191                 return 0;
192         }
193         child_pid[1] = ret;
194
```

```
195         if ((ret = fork()) == -1) {
196                 perror("fork 2");
197                 return errno;
198         }
199         else if (!ret) {
200                 // CHILD 5
201                 // can see pids of {3}
202                 // child_pid = (int *)get_shared_arr();
203
204                 if ((ret = fork()) == -1) {
205                         perror("fork 3");
206                         return errno;
207                 }
208                 else if (!ret) {
209                         // CHILD 3
210                         // can see pids of {4}
211
212                         child_pid[3] = getpid();
213
214                         if (setpgid(child_pid[3], child_pid[3]) == -1) {
215                                 perror("setpgid(3, 3");
216                                 return errno;
217                         }
218
219                         if (sem_post(pgid_sem) == -1) {
220                                 perror("setpgid(3, 3) done synchronization");
221                                 return errno;
222                         }
223
224                         // NOWWWWW tell 5  that pid of 3 is available in shared memory, and it can call setpgid safely
225                         if (sem_post(fork_sem) == -1) {
226                                 perror("P operation in 3");
227                                 return errno;
228                         }
229
230
231                         if ((ret = fork()) == -1) {
232                                 perror("fork 4");
233                                 return errno;
234                         }
235                         else if (!ret) {
236                                 // CHILD 4
237                                 // can see pids of {}
238                                 // child_pid = (int *)get_shared_arr();
239
240                                 if (setpgid(0, child_pid[3]) == -1) {
241                                         perror("setpgid(4, 3)");
242                                         return errno;
243                                 }
244
245                                 // NOWWW tell 0 that 4 has moved to new process group
246
247                                 if (sem_post(pgid_sem) == -1) {
248                                         perror("setpgid(4, 3) done synchronization");
249                                         return errno;
250                                 }
251
252
253                                 // wait for 0 to allow exiting
254                                 if (sem_wait(exit_sem) == -1) {
255                                         perror("V operation in 4");
256                                         return errno;
257                                 }
258
259                                 printf("%d is child of %d\n", getpid(), getppid());
260
261                                 return 0;
```

```
                }

                child_pid[4] = ret;


                if (sem_wait(exit_sem) == -1) {
                        perror("V operation in 3");
                        return errno;
                }

                printf("%d is child of %d\n", getpid(), getppid());

                examine_child(child_pid[4]);

                return 0;
        }
        child_pid[3] = ret;


        // wait for 0 to allow exiting
        if (sem_wait(exit_sem) == -1) {
                perror("V operation in 2");
                return errno;
        }

        printf("%d is child of %d\n", getpid(), getppid());

        examine_child(child_pid[3]);

        return 0;
}
child_pid[2] = ret;

// wait till all processes created, and the 3 setpgid calls finish
int i;
for (i = 0; i < 3; i++) {
        if (sem_wait(pgid_sem) == -1) {
                perror("V operation in 0");
                return errno;
        }
}

child_pid[0] = getpid();

if (kill(-child_pid[3], SIGUSR1) == -1) {
        perror("kill");
        return errno;
}

for (i = 0; i <= 5; i++) {
        printf("Process %d = %d\n", i, child_pid[i]);
}


for (i = 0; i < 5; i++) {
        if (sem_post(exit_sem) == -1) {
                perror("P by 0");
                return errno;
        }
}

examine_child(child_pid[1]);
examine_child(child_pid[2]);

return 0;
```

```
}
```

Explanation

- Shared memory is created for sharing pids and semaphores
- NO. of Semaphores used = 3
- *fork_sem* is used by process 3 to tell process 5 that it's pid is available in shared memory, and that it's process group has been created
- *pgid_sem* is used for telling 0 that all 3 setpgid calls are done
- *exit_sem* is used for telling 1-5 that it has called kill, and they are free to exit now

Output

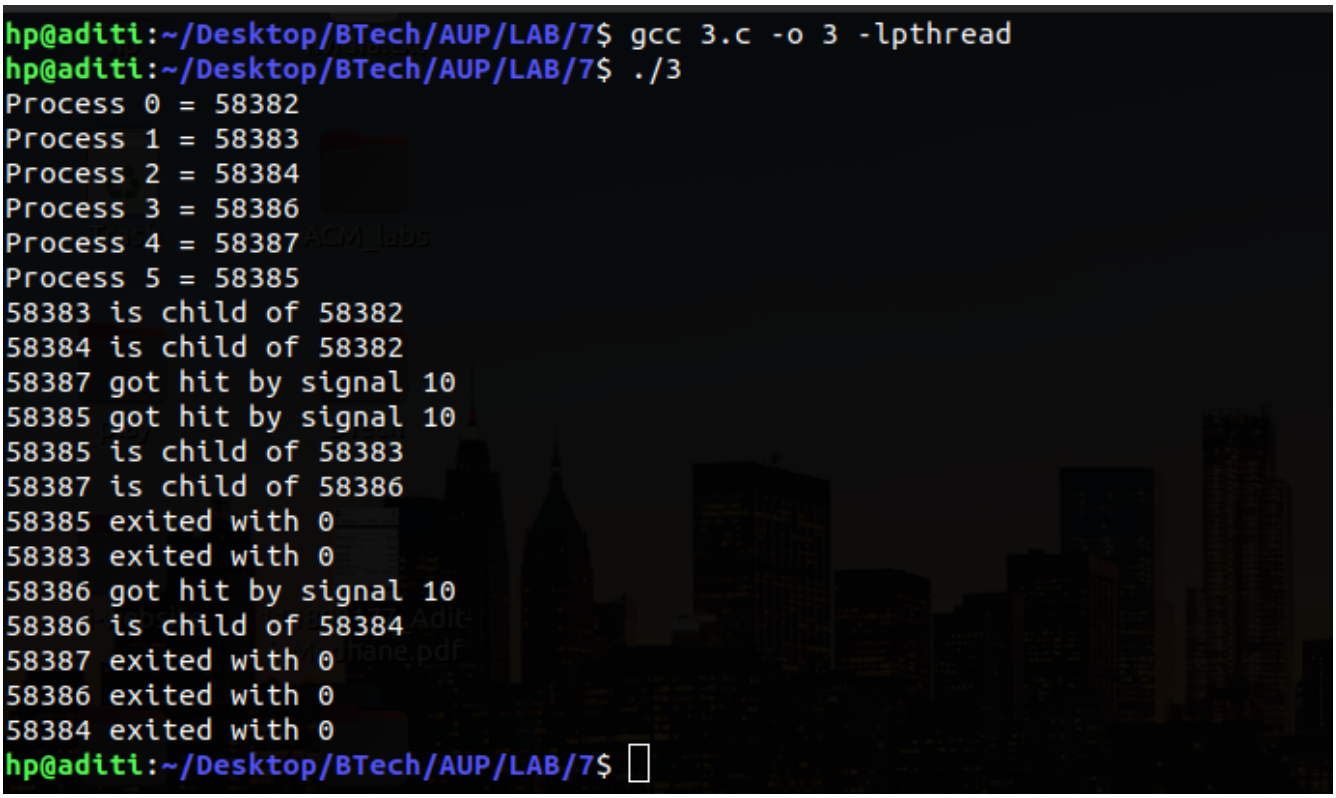


Figure 2: Execution of 3, SIGUSR hits 3, 4, 5