

# AUP Assignment 7

111703013 Akshay Rajesh Deodhar

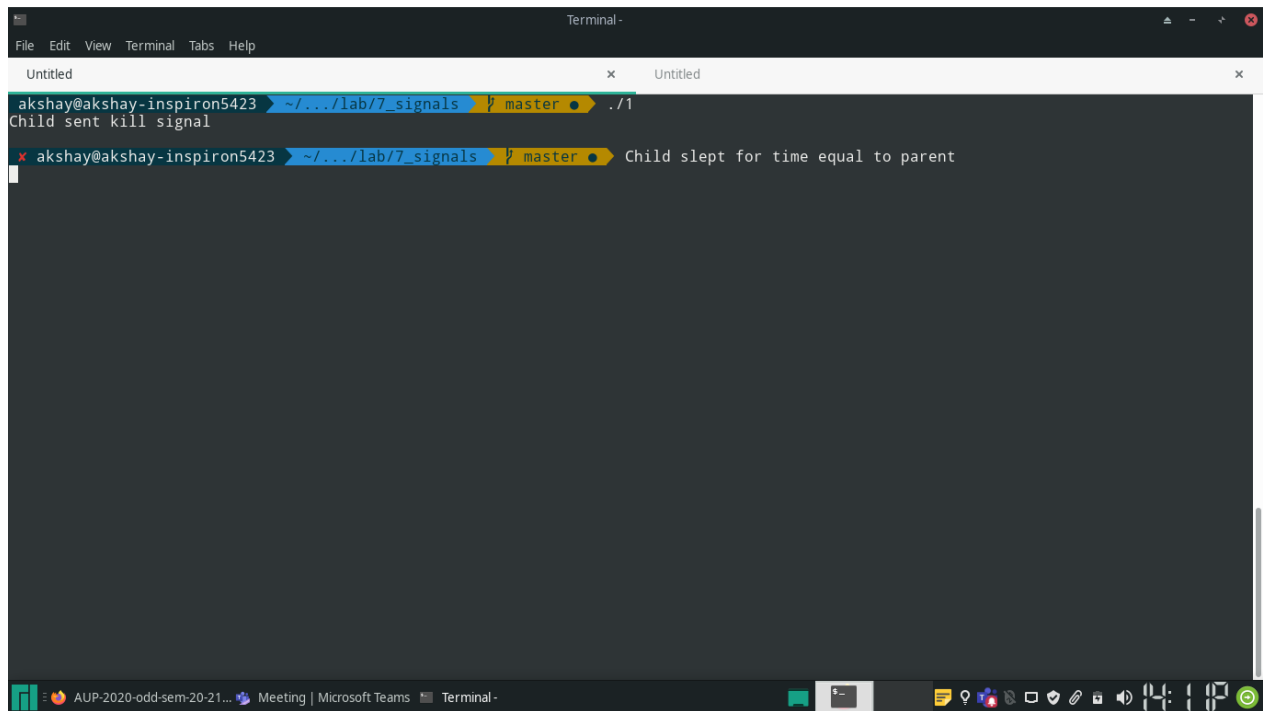
11th October 2020

**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

```
1
2
3  #include <sys/types.h>
4  #include <unistd.h>
5  #include <signal.h>
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <errno.h>
9
10 int main(void) {
11     int child_pid;
12
13     if ((child_pid = fork()) == -1) {
14         /* fork failed */
15         perror("fork");
16         return errno;
17     }
18     else if (child_pid) {
19         /* parent */
20         sleep(5);
21         printf("Parent is Alive\n");
22     }
23     else {
24         /* child */
25         if (kill(getppid(), SIGINT) == -1) {
26             perror("SIGINT to parent");
27             return errno;
28         }
29         printf("Child sent kill signal\n");
30         sleep(5);
31         printf("Child slept for time equal to parent\n");
32     }
33
34     return 0;
35 }
```

## Output



```
Terminal -
File Edit View Terminal Tabs Help
Untitled x Untitled
akshay@akshay-inspiron5423 ~/.../lab/7_signals master • ./1
Child sent kill signal
x akshay@akshay-inspiron5423 ~/.../lab/7_signals master • Child slept for time equal to parent
```

Figure 1: Parent is alive not printed, parent killed by SIGINT sent by child

## 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:

- Normal way executing in the foreground
- Run as a back ground process and send SIGCHLD to it from the shell

## Code

```
1
2
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6 #include <signal.h>
7 #include <errno.h>
8 #include <stdio.h>
9
10 void print_sigchld_msg(int sig_number) {
11     printf("SIGCHLD hits parent (msg)\n");
12 }
13
14 int main(void) {
15     pid_t child_id;
16
```

```

17     int dead_child;
18     int status;
19
20     /* set handler for SIGCHLD */
21     signal(SIGCHLD, print_sigchld_msg);
22
23     if ((child_id = fork()) == -1) {
24         perror("fork");
25         return errno;
26     }
27     else if (child_id) {
28         /* parent */
29
30         if ((dead_child = wait(&status)) == -1) {
31             perror("wait failed");
32             return errno;
33         }
34         else {
35             printf("Wait returned, child %d exited\n", dead_child);
36         }
37     }
38     else {
39         /* Child */
40         printf("Child sleeping\n");
41         sleep(10);
42     }
43
44     return 0;
45 }
46

```

## Output

```
Terminal - aup@akshay-inspiron5423:/home/akshay/Desktop/COEP/BTech/AUP/assignments/lab/6_pctl_even_more
File Edit View Terminal Tabs Help
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ● ./2
Child sleeping
SIGCHLD hits parent (msg)
Wait returned, child 10405 exited
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ●
```

Figure 2: Normal way of executing the program, SIGCHLD hits once

```
Terminal - aup@akshay-inspiron5423:/home/akshay/Desktop/COEP/BTech/AUP/assignments/lab/6_pctl_even_more
File Edit View Terminal Tabs Help
aup@akshay-inspiron5423:/home/akshay/Desktop/COEP/BTech/AUP/assignments/lab/6_pctl_even_more x Untitled x
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ● ./2
Child sleeping
SIGCHLD hits parent (msg)
Wait returned, child 13917 exited
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ● ./2 &
[1] 13967
Child sleeping
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ● ps
  PID TTY          TIME CMD
  6812 pts/1    00:00:01 bash
 13967 pts/1    00:00:00 2
 13970 pts/1    00:00:00 2
 14020 pts/1    00:00:00 ps
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ● kill -17 13967
SIGCHLD hits parent (msg)
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ● SIGCHLD hits parent (msg)
Wait returned, child 13970 exited
```

Figure 3: Run as a back ground process and send SIGCHLD to it from the shell, SIGCHLD hits twice

**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.

- 1 and 2 are children of 0
- 5 is child of 1
- 3 is child of 2
- 4 is child of 3

## Code

```

1
2 #include <sys/types.h>
3 #include <sys/mman.h>
4 #include <sys/stat.h>
5 #include <sys/wait.h>
6 #include <unistd.h>
7 #include <fcntl.h>
8 #include <semaphore.h>
9 #include <signal.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <errno.h>
13
14 #define SHARED_ARR "shared_array"
15 #define SEMAPHORE_FORKING "fork_semaphore"
16 #define N 6
17
18 /* to ensure that fp can be used by all processes */
19 static int fp;
20
21 void print_message(int signo) {
22     printf("%d got hit by signal %d\n", getpid(), signo);
23 }
24
25 void examine_child(int pid) {
26     int status;
27
28     if (waitpid(pid, &status, 0) == -1) {
29         perror("wait");
30         exit(errno);
31     }
32
33     if (WIFEXITED(status)) {
34         printf("%d exited with %d\n", pid, WEXITSTATUS(status));
35     }
36     else if (WIFSIGNALED(status)) {
37         printf("%d killed by signal %d\n", pid, WTERMSIG(status));
38     }
39     else {
40         printf("%d died in some other way\n", pid);
41     }
42 }
43
44 void *get_shared_arr() {
45     void *buf;
46     /* assert: fp is a file descriptor which points to the shared memory,
47     * and is inherited by all processes */

```

```

48     if ((buf = mmap(NULL, sizeof(int) * N, PROT_READ | PROT_WRITE, MAP_SHARED, fp, 0)) == MAP_FAILED) {
49         perror("shared memory mmap() failed");
50         exit(errno);
51     }
52     return buf;
53 }
54
55
56 int main(void) {
57
58     /* process 0 */
59     /* can see the pids of {1, 2} */
60     sem_t *fork_sem, *pgid_sem, *exit_sem;
61     int *child_pid;
62     int ret;
63
64     if (signal(SIGUSR1, print_message) == SIG_ERR) {
65         perror("SIGUSR1");
66         return errno;
67     }
68
69     /* link shared memory address to process- this will be visible in
70      * children */
71     if ((child_pid= mmap(NULL, sizeof(int) * N,
72                          PROT_READ | PROT_WRITE,
73                          MAP_SHARED | MAP_ANONYMOUS,
74                          -1, 0)) == MAP_FAILED) {
75
76         perror("mmap");
77         return errno;
78     }
79
80     /* create a shared memory map for semaphore */
81     if ((fork_sem = mmap(NULL, sizeof(sem_t),
82                          PROT_READ | PROT_WRITE,
83                          MAP_SHARED | MAP_ANONYMOUS,
84                          -1, 0)) == MAP_FAILED) {
85
86         perror("semaphore");
87         return errno;
88     }
89     /* initialize semaphore with initial value 0- when 5 and 3 get
90      * created, this is incremented by 1 each. 0 will synchronize by calling
91      * down() on this twice */
92     if (sem_init(fork_sem, 1, 0) == -1) {
93         perror("semaphore initialization");
94         return errno;
95     }
96
97     /* create a shared memory map for semaphore */
98     if ((exit_sem = mmap(NULL, sizeof(sem_t),
99                          PROT_READ | PROT_WRITE,
100                          MAP_SHARED | MAP_ANONYMOUS,
101                          -1, 0)) == MAP_FAILED) {
102
103         perror("semaphore");
104         return errno;
105     }
106

```

```

107      /* initialize semaphore with initial value 0- when signals have been
108      * sent, this will be incremented 5 times, wherein all processes will
109      * exit */
110      if (sem_init(&exit_sem, 1, 0) == -1) {
111          perror("semaphore initialization");
112          return errno;
113      }
114
115      /* create a shared memory map for semaphore */
116      if ((pgid_sem = mmap(NULL, sizeof(sem_t),
117                          PROT_READ | PROT_WRITE,
118                          MAP_SHARED | MAP_ANONYMOUS,
119                          -1, 0)) == MAP_FAILED) {
120
121          perror("semaphore");
122          return errno;
123      }
124
125      /* initialize semaphore with initial value 0- when all processes have
126      * called setpgid, the parent can send signals */
127      if (sem_init(&pgid_sem, 1, 0) == -1) {
128          perror("semaphore initialization");
129          return errno;
130      }
131
132      /* assert: now each child will have access to the semaphore, unless the
133      * memory region is purposely unliked */
134
135
136      if ((ret = fork()) == -1) {
137          perror("fork 1");
138          return errno;
139      }
140      else if (!ret) {
141          /* child 1 */
142          /* can see pids of {5} */
143          /* child_pid = (int *)get_shared_arr(); */
144
145          if ((ret = fork()) == -1) {
146              perror("fork 5");
147              return errno;
148          }
149          else if (!ret) {
150              /* child 5 */
151              /* can see pids of {} */
152              /* child_pid = (int *)get_shared_arr(); */
153
154              /* wait for the PID of 3 to be available */
155              if (sem_wait(&fork_sem) == -1) {
156                  perror("P operation in 5");
157              }
158
159              /* set own process group to 3 */
160              if (setpgid(0, child_pid[3]) == -1) {
161                  perror("setpgid(5, 3)");
162                  return errno;
163              }
164
165              if (sem_post(&pgid_sem) == -1) {

```

```

166         perror("setpgid(5, 3) done synchronization");
167         return errno;
168     }
169
170
171     /* wait for 0 to allow exiting */
172     if (sem_wait(exit_sem) == -1) {
173         perror("V operation in 4");
174         return errno;
175     }
176
177     printf("%d is child of %d\n", getpid(), getppid());
178
179     return 0;
180 }
181 child_pid[5] = ret;
182
183
184
185     /* wait for 0 to allow exiting */
186     if (sem_wait(exit_sem) == -1) {
187         perror("V operation in 2");
188         return errno;
189     }
190
191     printf("%d is child of %d\n", getpid(), getppid());
192
193     examine_child(child_pid[5]);
194
195     return 0;
196 }
197 child_pid[1] = ret;
198
199 if ((ret = fork()) == -1) {
200     perror("fork 2");
201     return errno;
202 }
203 else if (!ret) {
204     /* child 2 */
205     /* can see pids of {3} */
206     /* child_pid = (int *)get_shared_arr(); */
207
208     if ((ret = fork()) == -1) {
209         perror("fork 3");
210         return errno;
211     }
212     else if (!ret) {
213         /* child 3 */
214         /* can see pids of {4} */
215
216         child_pid[3] = getpid();
217
218         if (setpgid(child_pid[3], child_pid[3]) == -1) {
219             perror("setpgid(3, 3)");
220             return errno;
221         }
222
223         if (sem_post(pgid_sem) == -1) {
224             perror("setpgid(3, 3) done synchronization");

```



```

225         return errno;
226     }
227
228     /* tell 5 that pid of 3 is available in shared memory,
229     * and it can call setpgid safely*/
230     if (sem_post(fork_sem) == -1) {
231         perror("P operation in 3");
232         return errno;
233     }
234
235
236     if ((ret = fork()) == -1) {
237         perror("fork 4");
238         return errno;
239     }
240     else if (!ret) {
241         /* child 4*/
242         /* can see pids of {} */
243         /* child_pid = (int *)get_shared_arr(); */
244
245         if (setpgid(0, child_pid[3]) == -1) {
246             perror("setpgid(4, 3)");
247             return errno;
248         }
249
250         /* tell 0 that 4 has moved to new process group
251         * */
252         if (sem_post(pgid_sem) == -1) {
253             perror("setpgid(4, 3) done synchronization");
254             return errno;
255         }
256
257
258         /* wait for 0 to allow exiting */
259         if (sem_wait(exit_sem) == -1) {
260             perror("V operation in 4");
261             return errno;
262         }
263
264         printf("%d is child of %d\n", getpid(), getppid());
265
266         return 0;
267     }
268
269     child_pid[4] = ret;
270
271
272
273     if (sem_wait(exit_sem) == -1) {
274         perror("V operation in 3");
275         return errno;
276     }
277
278     printf("%d is child of %d\n", getpid(), getppid());
279
280     examine_child(child_pid[4]);
281
282     return 0;
283 }

```

```

284         child_pid[3] = ret;
285
286
287         /* wait for 0 to allow exiting */
288         if (sem_wait(exit_sem) == -1) {
289             perror("V operation in 2");
290             return errno;
291         }
292
293         printf("%d is child of %d\n", getpid(), getppid());
294
295         examine_child(child_pid[3]);
296
297         return 0;
298     }
299     child_pid[2] = ret;
300
301     /* wait till all processes created, and the 3 setpgid calls finish */
302     int i;
303     for (i = 0; i < 3; i++) {
304         if (sem_wait(pgid_sem) == -1) {
305             perror("V operation in 0");
306             return errno;
307         }
308     }
309
310     child_pid[0] = getpid();
311
312     if (kill(-child_pid[3], SIGUSR1) == -1) {
313         perror("kill");
314         return errno;
315     }
316
317     for (i = 0; i <= 5; i++) {
318         printf("Process %d = %d\n", i, child_pid[i]);
319     }
320
321
322     for (i = 0; i < 5; i++) {
323         if (sem_post(exit_sem) == -1) {
324             perror("P by 0");
325             return errno;
326         }
327     }
328
329     examine_child(child_pid[1]);
330     examine_child(child_pid[2]);
331
332     return 0;
333 }
334 }

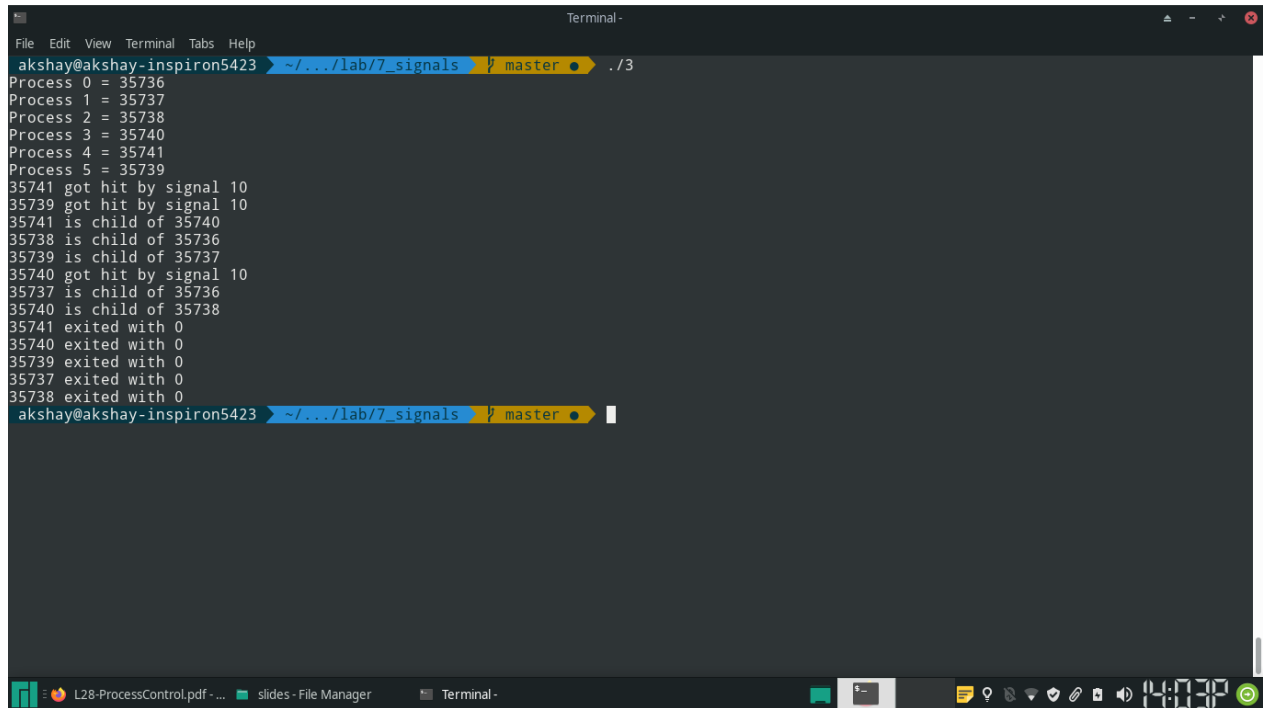
```

## Explanation

- Shared memory is created for sharing pids and semaphores
- 3 semaphores are used
- *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



```
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ● ./3
Process 0 = 35736
Process 1 = 35737
Process 2 = 35738
Process 3 = 35740
Process 4 = 35741
Process 5 = 35739
35741 got hit by signal 10
35739 got hit by signal 10
35741 is child of 35740
35738 is child of 35736
35739 is child of 35737
35740 got hit by signal 10
35737 is child of 35736
35740 is child of 35738
35741 exited with 0
35740 exited with 0
35739 exited with 0
35737 exited with 0
35738 exited with 0
akshay@akshay-inspiron5423 ~/.../lab/7_signals master ●
```

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