

## one\_to\_seven.c

```
1 // Assignment 1-a
2 // 1. Include necessary header files for system calls and standard I/O.
3 // 2. Define the main function.
4 // 3. Declare variables for process IDs and counters for child and grandchild processes.
5 // 4. Call fork() to create a child process.
6 //     a. If fork() fails, print an error message and exit with status 1.
7 //     b. If fork() returns 0 (child process):
8 //         i. Increment the child process counter.
9 //         ii. Print the PID and PPID of the child process.
10 //            iii. Call fork() again to create a grandchild process.
11 //                 - If fork() fails, print an error message and exit with status 1.
12 //                 - If fork() returns 0 (grandchild process):
13 //                     * Increment the grandchild process counter.
14 //                     * Print the PID, PPID, and roll number.
15 //                     * Exit with status 0.
16 //                 - Wait for the grandchild process to complete.
17 //                 - Exit the child process with status 0.
18 //            c. If fork() returns a positive value (parent process):
19 //                i. Print the PID of the parent process.
20 //                ii. Wait for the child process to complete.
21 //                iii. Print the total number of child and grandchild processes created.
22 // 5. Return 0 to indicate successful execution.
23
24 #include <stdio.h>
25 #include <stdlib.h>
26 #include <unistd.h>
27 #include <sys/types.h>
28 #include <sys/wait.h> // Include this for wait()
29
30 int main() {
31     pid_t pid, gcid;
32     int child_count = 1, grandchild_count = 1;
33
34     pid = fork();
35     if (pid < 0) {
36         perror("Fork failed");
37         exit(1);
38     }
39
40     if (pid == 0) { // Child process
41         child_count++;
42         printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
43
44         gcid = fork();
45         if (gcid < 0) {
46             perror("Fork failed");
47             exit(1);
48         }
49
50         if (gcid == 0) { // Grandchild process
51             grandchild_count++;
52             printf("Grandchild Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
53             printf("Roll Number: CSM24004\n");
54     }
```

```

54         exit(0);
55     }
56
57     wait(NULL); // Ensure grandchild completes
58     exit(0);
59 }
60 else { // Parent process
61     printf("Parent Process: PID = %d\n", getpid());
62     wait(NULL); // Ensure child completes
63     printf("Total Child Processes Created: %d\n", child_count);
64     printf("Total Grandchild Processes Created: %d\n", grandchild_count);
65 }
66 return 0;
67 }
68
69
70 // Assignment 1-b
71 // Function to print logged-in users
72 // - Open utmp file
73 // - Loop through entries
74 // - If entry is a user process, print user details
75 // - Close utmp file
76
77 // Main function
78 // - Declare process IDs
79 // - Fork a child process
80 //   - If fork fails, print error and exit
81 //   - If in child process:
82 //     - Print child process details
83 //     - Fork a grandchild process
84 //       - If fork fails, print error and exit
85 //       - If in grandchild process:
86 //         - Print grandchild process details
87 //         - Call function to print logged-in users
88 //         - Exit grandchild process
89 //       - Wait for grandchild to finish
90 //     - Exit child process
91 //   - If in parent process:
92 //     - Print parent process details
93 //     - Wait for child to finish
94 //   - Print message indicating child process finished
95
96 #include <stdio.h>
97 #include <stdlib.h>
98 #include <unistd.h>
99 #include <sys/types.h>
100 #include <sys/wait.h>
101 #include <utmp.h>
102
103 void print_logged_in_users() {
104     struct utmp *entry;
105     setutent(); // Open /var/run/utmp
106     while ((entry = getutent()) != NULL) {
107         if (entry->ut_type == USER_PROCESS) { // Only print active user logins
108             printf("User: %s, Terminal: %s, Host: %s\n", entry->ut_user, entry->ut_line, entry-
>ut_host);

```

```
109         }
110     }
111     endutent(); // Close file
112 }
113
114 int main() {
115     pid_t pid, gcid;
116
117     pid = fork();
118     if (pid < 0) {
119         perror("Fork failed");
120         exit(1);
121     }
122
123     if (pid == 0) { // Child process
124         printf("Child Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
125
126         gcid = fork();
127         if (gcid < 0) {
128             perror("Fork failed");
129             exit(1);
130         }
131
132         if (gcid == 0) { // Grandchild process
133             printf("Grandchild Process: PID = %d, Parent PID = %d\n", getpid(), getppid());
134             print_logged_in_users();
135             exit(0);
136         }
137
138         wait(NULL); // Child waits for grandchild
139         exit(0);
140     }
141     else { // Parent process
142         printf("Parent Process: PID = %d\n", getpid());
143         wait(NULL); // Parent waits for child
144         printf("Parent: Child process finished execution.\n");
145     }
146
147     return 0;
148 }
149
150 // Assignment 1-c: Child_program.c
151 // Pseudo code for child program:
152 // 1. Include necessary headers.
153 // 2. Define the main function with arguments for argc and argv.
154 // 3. Print the program name using argv[0].
155 // 4. Print the PID of the child process using getpid().
156 // 5. Exit the program.
157
158 #include <stdio.h>
159 #include <unistd.h>
160
161 int main(int argc, char *argv[]) {
162     printf("Child Program Name: %s\n", argv[0]);
163     printf("Child Process is running with PID: %d\n", getpid());
164 }
```

```
165     return 0;
166 }
167
168 // Assignment 1-d: Parent_program.c
169 // Pseudo code for the parent program:
170 // 1. Create a child process using fork().
171 // 2. In the child process:
172 //     a. Print the PID and parent PID.
173 //     b. Prepare arguments for execv().
174 //     c. Overlay the child process with a new program using execv().
175 //     d. Handle errors if execv() fails.
176 // 3. In the parent process:
177 //     a. Wait for the child process to finish using wait().
178 //     b. Check if the child exited normally using WIFEXITED.
179 //     c. Print the child's exit status or an error message.
180 //     d. Print the parent's PID.
181
182 #include <unistd.h>
183 #include <sys/types.h>
184 #include <sys/wait.h>
185
186 int main() {
187     pid_t pid;
188     int status;
189
190     pid = fork();
191
192     if (pid < 0) {
193         perror("Fork failed");
194         exit(1);
195
196     } else if (pid == 0) {
197         // CP
198         printf("Child Process before execv(): PID = %d, Parent PID = %d\n", getpid(), getppid());
199
200         // Prepare arguments for execv()
201         char *args[] = {"./child_program", NULL};
202
203         //Overlay child process with child_program using execv()
204         if (execv(args[0], args) < 0) {
205             perror("execv failed");
206             exit(1);
207         }
208     } else {
209         //Parent process
210         //wait for child process to finish
211         if (wait(&status) < 0) {
212             perror("wait failed");
213             exit(1);
214         }
215
216         // Check if child exited normally
217         if (WIFEXITED(status)) {
218             printf("Parent Process: Child exited with status = %d\n", WEXITSTATUS(status));
219         } else {
220             printf("Parent Process: Child did not exit normally\n");
```

```
221     }
222
223     printf("Parent Process: PID = %d\n", getpid());
224 }
225
226     return 0;
227 }
228
229 // Assignment 2-a:
230 /*
231 * Assignment 2(a)
232 * 1. Include necessary headers for input/output, process control, and signal handling.
233 * 2. Define a signal handler function:
234 *     a. Print a message indicating SIGINT was received.
235 *     b. Use fork() to create a child process.
236 *     c. In the child process:
237 *         i. Print the child process ID and its parent process ID.
238 *         ii. Exit the child process to prevent it from continuing the parent's execution flow.
239 *     d. In the parent process:
240 *         i. Use wait() to ensure the child process completes execution to avoid zombie processes.
241 * 3. In the main function:
242 *     a. Register the signal handler for SIGINT using signal().
243 *     b. Use an infinite loop with pause() to keep the process alive and waiting for signals.
244 * 4. Compile and run the program. Press Ctrl+C to test the signal handling and process creation.
245 */
246 /*
247 * This program demonstrates handling SIGINT to create a child process using fork().
248 * When Ctrl+c is pressed, the parent and child processes print their PIDs and PPIDs.
249 */
250
251 #include <stdio.h>
252 #include <unistd.h>
253 #include <wait.h>
254 #include <stdlib.h>
255 #include <signal.h>
256
257 void signal_handler(int num) {
258     printf("Ctrl + C pressed\n");
259     pid_t pid;
260     pid = fork();
261     if (pid < 0) {
262         perror("Fork Failed");
263         exit(1);
264     }
265     if (pid == 0) {
266         printf("Child Process Created \nPID = %d, Parent PID = %d\n", getpid(),
267 getppid());
268         exit(0); // Terminates child immediately
269     }
270
271     else {
272         wait(NULL); //Ensure child completes to prevent to become zombiee
273     }
274
275 int main() {
```

```
276     signal(SIGINT, signal_handler);
277
278     // Keep process alive to handle signal
279     while(1){
280         pause(); // No busy loop
281     }
282     return 0;
283 }
284
285
286 // Assignment 2-b:
287 /*
288 * Assignment 2(b)
289 * 1. Include necessary headers for input/output, signal handling, and process control.
290 * 2. Define a signal handler function to display the received signal ID.
291 * 3. In the main function:
292 *     a. Declare variables for Process ID (pid) and Signal ID (sig).
293 *     b. Register the signal handler for all possible signals (1 to 31).
294 *     c. Prompt the user to enter the Process ID and Signal ID.
295 *     d. Validate the input to ensure the PID and Signal ID are valid.
296 *     e. Use the kill() system call to send the specified signal to the given process.
297 *     f. Handle errors if the kill() system call fails.
298 * 4. Display appropriate messages for successful or failed signal delivery.
299 * 5. Exit the program.
300 */
301 /*
302 * This program takes a Process ID (PID) and a Signal ID as input and sends the signal to the
303 * specified process.
304 * The purpose is to demonstrate the use of the kill() system call.
305 * The receiving process (modified Assignment 2(a)) should handle the signal and display the
306 * received signal ID.
307 */
308
309 #include <stdio.h>
310 #include <stdlib.h>
311 #include <signal.h>
312 #include <unistd.h>
313
314 // Signal handler function to display received signal ID
315 void signal_handler(int signum) {
316     printf("Received Signal: %d\n", signum);
317 }
318
319 int main() {
320     pid_t pid;
321     int sig;
322
323     // All possible signals
324     for (int i = 1; i < 32; i++) {
325         signal(i, signal_handler);
326     }
327     printf("Enter Process ID: ");
328     scanf("%d", &pid);
329     printf("Enter Signal ID: ");
330     scanf("%d", &sig);
```

```

330     // Validate input
331     if (pid <= 0 || sig <= 0 || sig > 31) {
332         printf("Invalid PID or Signal ID\n");
333         return 1;
334     }
335
336     // Send signal to process using kill()
337     if (kill(pid, sig) == 0) {
338         printf("Signal %d sent to process %d successfully.\n", sig, pid);
339     } else {
340         perror("Failed to send signal");
341         return 1;
342     }
343
344     return 0;
345 }
346
347 // Assignment 2-c
348 /*
349 * Assignment 2(c)
350 * 1. Include necessary headers for input/output and pthread library.
351 * 2. Define a thread function `print_hello` that prints "HELLO WORLD" and exits using
352 *    pthread_exit().
353 * 3. In the main function:
354 *      a. Declare a thread identifier of type `pthread_t`.
355 *      b. Use `pthread_create()` to create a new thread that runs the `print_hello` function.
356 *          - Check for errors in thread creation and handle them using `perror()` and `exit()` .
357 *      c. Use `pthread_join()` to wait for the created thread to finish execution.
358 *      d. Print a message indicating the thread has completed.
359 * 4. Return 0 to indicate successful program execution.
360 *
361 * Purpose of System Calls:
362 * - `pthread_create()`: Used to create a new thread at the user level.
363 * - `pthread_exit()`: Used to terminate the thread explicitly.
364 * - `pthread_join()`: Used to wait for the thread to complete before proceeding.
365 */
366
367 * This program creates a user-level thread to print "HELLO WORLD".
368 */
369 #include <stdio.h>
370 #include <stdlib.h>
371 #include <pthread.h>
372
373 // Thread function to print message
374 void* print_hello(void* arg) {
375     printf("HELLO WORLD\n");
376     pthread_exit(NULL);
377 }
378
379 int main()
380 {
381     pthread_t thread_id;
382     // Create a thread
383     if (pthread_create(&thread_id, NULL, print_hello, NULL) != 0) {
384         perror("pthread_create");
385         exit(1);

```

```
385     }
386
387     // Wait for the thread to finish
388     pthread_join(thread_id, NULL);
389     printf("Thread completed\n");
390
391     return 0;
392 }
393
394 // Assignment 3
395 // Include necessary header files for signal handling, process creation, and inter-process
396 // communication
397
398 // Define a signal handler function 'handler' to handle SIGINT signal
399 // Inside the handler:
400 // 1. Create a pipe for communication between parent and child processes
401 // 2. Use fork() to create a child process
402 // 3. In the child process:
403 //     a. Close the read end of the pipe
404 //     b. Write the child's process ID to the pipe
405 //     c. Take input N (number of Fibonacci terms) from the user
406 //     d. Write N to the pipe
407 //     e. Generate the Fibonacci series up to N terms and write each term to the pipe
408 //     f. Write the signal ID (SIGINT) to the pipe
409 //     g. Close the write end of the pipe and exit
410 // 4. In the parent process:
411 //     a. Close the write end of the pipe
412 //     b. Read the child's process ID from the pipe and display it
413 //     c. Read N from the pipe and display the Fibonacci series terms received from the child
414 //     d. Read the signal ID from the pipe and display it
415 //     e. Close the read end of the pipe and wait for the child process to terminate
416
417 // In the main function:
418 // 1. Register the signal handler for SIGINT
419 // 2. Enter an infinite loop to simulate a running program
420 // 3. Print a message indicating the main program is running and sleep for 1 second
421 #include <stdio.h>
422 #include <signal.h>
423 #include <unistd.h>
424 #include <sys/types.h>
425 #include <sys/wait.h>
426 #include <stdlib.h>
427
428 void handler(int sig) {
429     int pipefd[2];
430     if(pipe(pipefd) == -1) {
431         perror("pipe Failed");
432         exit(1);
433     }
434     pid_t pid = fork();
435     if(pid == -1) {
436         perror("fork Failed");
437         exit(1);
438     }
439     if(pid == 0) {
440         close(pipefd[0]);
```

```

440     pid_t mypid = getpid();
441     write(pipefd[1], &mypid, sizeof(pid_t));
442
443     int N;
444     printf("Enter N: ");
445     scanf("%d", &N);
446     write(pipefd[1], &N, sizeof(int));
447
448     int a = 0, b = 1, c;
449     for(int i = 0; i < N; i++) {
450         write(pipefd[1], &a, sizeof(int));
451         c = a + b;
452         a = b;
453         b = c;
454     }
455     write(pipefd[1], &sig, sizeof(int));
456     close(pipefd[1]);
457     exit(0);
458 }
459 else {
460     close(pipefd[1]);
461     pid_t child_pid;
462     read(pipefd[0], &child_pid, sizeof(pid_t));
463     printf("child PID: %d\n", child_pid);
464
465     int N;
466     read(pipefd[0], &N, sizeof(int));
467     printf("Fibonacci series up to %d terms: ", N);
468
469     int term;
470     for(int i = 0; i < N; i++) {
471         read(pipefd[0], &term, sizeof(int));
472         printf("%d ", term);
473     }
474     printf("\n");
475     int signal_id;
476     read(pipefd[0], &signal_id, sizeof(int));
477     printf("Signal ID: %d\n", signal_id);
478
479     close(pipefd[0]);
480     wait(NULL);
481 }
482 }
483
484 int main() {
485     signal(SIGINT, handler);
486     while(1) {
487         printf("Main program running... \n");
488         sleep(1);
489     }
490     return 0;
491 }
492
493 // Assignment 4
494 // This program demonstrates file locking using `fcntl()` in a multi-process environment.
495 // It locks a file, reads its contents, holds the lock for 10 seconds, and then releases the lock.

```

```

496 // 1. Include necessary headers for file operations, locking, and process control.
497 // 2. Define a function `lock_file` to set or release a file lock using `fcntl()`.
498 //     - Initialize a `struct flock` with appropriate lock type, offset, and length.
499 //     - Use `fcntl()` with `F_SETLKW` to set or release the lock, handling errors.
500 // 3. In the `main` function:
501 //     - Check if the filename is provided as a command-line argument.
502 //     - Open the file in read-write mode using `fopen()` .
503 //     - Handle errors if the file cannot be opened.
504 //     - Print the process ID and attempt to lock the file using `lock_file` .
505 //     - Read the file contents using `read()` and print them to the console.
506 //     - Handle errors during reading and release the lock if necessary.
507 //     - Hold the lock for 10 seconds to simulate exclusive access.
508 //     - Release the lock using `lock_file` and close the file.
509 //     - Exit the program.
510 #include <stdio.h>
511 #include <fcntl.h>
512 #include <unistd.h>
513 #include <stdlib.h>
514
515 void lock_file(int fd, short lock_type) {
516     struct flock lock;
517     lock.l_type = lock_type;      // F_WRLCK (write lock) or F_UNLCK (unlock)
518     lock.l_whence = SEEK_SET;    // Start from beginning of file
519     lock.l_start = 0;           // Offset 0
520     lock.l_len = 0;             // Lock entire file (0 means to end)
521
522     if (fcntl(fd, F_SETLKW, &lock) == -1) {
523         perror("Error setting lock");
524         exit(EXIT_FAILURE);
525     }
526 }
527
528 int main(int argc, char *argv[]) {
529     FILE* fptr;
530     char content[255];
531
532     if (argc < 2) {
533         fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
534         return 1;
535     }
536
537
538     fptr = fopen(argv[1], "r+"); // "r+" for locking
539     if (fptr == NULL) {
540         printf("File open Unsuccessful\n");
541         return 1;
542     }
543
544     printf("Process %d: Attempting to lock file...\n", getpid());
545     lock_file(fileno(fptr), F_WRLCK);
546     printf("Process %d: File locked. Reading contents...\n", getpid());
547
548     int fd = fileno(fptr);
549     ssize_t bytes_read;
550     while ((bytes_read = read(fd, content, sizeof(content) - 1)) > 0) {
551         content[bytes_read] = '\0';

```

```

552     printf("%s", content);
553 }
554 if (bytes_read == -1) {
555     perror("Error reading file");
556     lock_file(fd, F_UNLCK);
557     fclose(fp);
558     return 1;
559 }
560
561 printf("\nProcess %d: Holding lock for 10 seconds...%n", getpid());
562 sleep(10);
563
564 printf("Process %d: Unlocking file...%n", getpid());
565 lock_file(fd, F_UNLCK);
566
567 fclose(fp);
568 return 0;
569 }
570
571 // Assignment 5 -a (program1.c)
572 // This program demonstrates the use of named pipes (FIFOs) for inter-process communication.
573 // Pseudo code for the program:
574 // 1. Define the FIFO file path.
575 // 2. Create a named FIFO (if it doesn't already exist) using the mknod() system call.
576 // 3. Open the FIFO in read-only mode.
577 // 4. Read data from the FIFO written by another process (Program2).
578 // 5. Display the received data.
579 // 6. Close the FIFO file descriptor.
580 // 7. Delete the FIFO after use to clean up resources.
581 #include <unistd.h>
582 #include <stdio.h>
583
584 int main() {
585     char *fifo_name = "/tmp/myfifo";
586     int fd = open(fifo_name, O_RDONLY);
587     if (fd == -1) {
588         perror("Open failed");
589         return 1;
590     }
591
592     char buffer[100];
593     int bytesRead = read(fd, buffer, sizeof(buffer));
594     if (bytesRead > 0) {
595         buffer[bytesRead] = '\0';
596         printf("Reader Received: %s\n", buffer);
597     }
598     // printf("%ld",sizeof(buffer));
599     // printf("Reader received: %s\n", buffer);
600
601     close(fd);
602     unlink(fifo_name); // Deleting the fifo
603     return 0;
604 }
605
606 // Assignment 5 -a (program2.c)
607 // This program demonstrates the use of named pipes (FIFOs) for inter-process communication.

```

```
608 // Pseudo code for Program2:  
609  
610 /*  
611 1. Include necessary header files for file operations, FIFO handling, and standard I/O.  
612 2. Define the FIFO name (path) as a string constant.  
613 3. Create the FIFO using the mknod() system call with appropriate permissions (read/write for all  
users).  
614 4. Open the FIFO in write-only mode using the open() system call.  
   - If the open() call fails, print an error message and exit the program.  
615 5. Prompt the user to input a message.  
616 6. Read the user input using fgets() and remove the trailing newline character.  
617 7. Write the user input to the FIFO using the write() system call.  
618 8. Print a confirmation message indicating the message has been sent.  
619 9. Close the FIFO file descriptor using the close() system call.  
620 10. Exit the program.  
621 */  
622 #include <fcntl.h>  
623 #include <sys/stat.h>  
624 #include <unistd.h>  
625 #include <string.h>  
626 #include <stdio.h>  
627  
628  
629 int main() {  
630     char *fifo_name = "/tmp/myfifo";  
631     mknod(fifo_name, S_IFIFO | 0666, 0); // Create FIFO  
632  
633     int fd = open(fifo_name, O_WRONLY);  
634     if (fd == -1) {  
635         perror("open failed");  
636         return 1;  
637     }  
638  
639     printf("Send your message from here!\nType: ");  
640     char message[100];  
641  
642     fgets(message, sizeof(message), stdin);  
643     //scanf("%s", message);  
644     message[strcspn(message, "\n")] = '\0'; // Remove newline character  
645  
646     write(fd, message, strlen(message) + 1);  
647     printf("Writer sent: %s\n", message);  
648  
649     close(fd);  
650     return 0;  
651 }  
652  
653 // Assignment 5 -b (client.c)  
654 // Include necessary headers  
655 // Define constants and message structure  
656  
657 // Main function  
658 // 1. Generate a unique key using ftok() to identify the message queue  
659 // 2. Connect to the message queue using msgget()  
660 // 3. Prompt the user to enter a message  
661 // 4. Remove the newline character from the input  
662 // 5. Set the message type to 1 (for server)
```

```
663 // 6. Send the message to the server using msgsnd()
664 // 7. Handle errors if message sending fails
665 // 8. Wait for a reply from the server (type 2) using msgrecv()
666 // 9. Handle errors if message receiving fails
667 // 10. Print the server's reply
668 // 11. Exit the program
669
670 // Summary:
671 // This client program demonstrates inter-process communication (IPC) using System V Message
672 // Queues.
673 // It connects to a message queue, sends a message to the server, and waits for a reply. The
674 // message
675 // queue ID is generated using ftok() and connected using msgget(). Messages are sent and received
676 // using msgsnd() and msgrecv(), respectively.
677 #include <stdio.h>
678 #include <stdlib.h>
679 #include <string.h>
680 #include <sys/types.h>
681 #include <sys/msg.h>
682
683 // Define MAX_SIZE
684 #define MAX_SIZE 100
685
686 // Message structure
687 struct msgbuf {
688     long mtype;
689     char mtext[MAX_SIZE];
690 };
691
692 int main() {
693     key_t key;
694     int msqid;
695     struct msgbuf message;
696
697     // Generate the same key as server
698     key = ftok("progfile", 65);
699
700     // Connect to the message queue
701     msqid = msgget(key, 0666);
702
703     // Get user input
704     printf("Client: enter a message: ");
705     fgets(message.mtext, MAX_SIZE, stdin);
706     message.mtext[strcspn(message.mtext, "\n")] = 0; // Remove newline
707     message.mtype = 1; // Type 1 for server
708
709     // Send message
710     if (msgsnd(msqid, &message, strlen(message.mtext) + 1, 0) == -1) {
711         perror("msgsnd failed");
712         exit(1);
713     }
714
715     // Receive reply (type 2 from server)
716     if (msgrecv(msqid, &message, MAX_SIZE, 2, 0) == -1) {
717         perror("msgrecv failed");
718         exit(1);
719     }
720 }
```

```
717     printf("Client: Server replied: %s\n", message.mtext);
718     return 0;
719 }
720
721 // Assignment 5 -b (server.c)
722 /*
723 * 1. Include necessary headers and define constants.
724 * 2. Define a structure for the message queue with a type and text field.
725 * 3. In the main function:
726 *    a. Generate a unique key using ftok().
727 *    b. Create a message queue using msgget() with appropriate permissions.
728 *    c. Print the message queue ID to stderr for debugging purposes.
729 *    d. Enter an infinite loop to handle messages:
730 *        i. Receive a message of type 1 from the client using msgrcv().
731 *        ii. Print the received message to the console.
732 *        iii. Prepare a reply message with type 2 and a predefined response.
733 *        iv. Send the reply message back to the client using msgsnd().
734 *        v. Break the loop after one message for demonstration purposes.
735 *    e. Remove the message queue using msgctl() with IPC_RMID to clean up.
736 *    f. Print a message indicating the server is exiting.
737 * 4. Return 0 to indicate successful execution.
738 */
739
740 * Summary:
741 * The server program demonstrates inter-process communication (IPC) using System V Message Queues.
742 * It receives a message from a client, processes it, sends a reply, and cleans up the message
queue.
743 */
744 // Server Program
745 #include <stdio.h>
746 #include <stdlib.h>
747 #include <string.h>
748 #include <sys/types.h>
749 #include <sys/ipc.h>
750 #include <sys/msg.h>
751
752 #define MAX_SIZE 100
753
754 // Message structuree
755 struct msgbuf {
756     long mtype; // Message type
757     char mtext[MAX_SIZE];
758 };
759
760 int main() {
761     key_t key;
762     int msqid;
763     struct msgbuf message;
764
765     // Generate a unique key
766     key = ftok("progfile", 65); // progfile is dummy file, 65 id a random character
767
768     // Create message queue
769     msqid = msgget(key, 0666 | IPC_CREAT);
770
771     // Print queue ID to stderr
```

```

772     fprintf(stderr, "Message Queue ID: %d\n", msqid);
773     printf("Server: Waiting for messages...\n");
774
775     while (1) {
776         // Receive message of type 1 (from client)
777         if (msgrcv(msqid, &message, MAX_SIZE, 1, 0) == -1) {
778             perror("msgrcv failed");
779             exit(1);
780         }
781
782         printf("Server: Received: %s\n", message.mtext);
783
784         // Prepare reply
785         message.mtype = 2; // Type 2 for client
786         strcpy(message.mtext, "Got it!");
787         if (msgsnd(msqid, &message, strlen(message.mtext) + 1, 0) == -1) {
788             perror("msgsnd failed");
789             exit(1);
790         }
791
792         // For demo, break after one message (remove this for continuous run)
793         break;
794     }
795
796     // Remove the message queue
797     if (msgctl(msqid, IPC_RMID, NULL) == -1) {
798         perror("msgctl failed");
799         exit(1);
800     }
801
802     printf("Server: Queue removed. Exiting.\n");
803     return 0;
804 }
805
806 // Assignment 6 -a (program1.c)
807 /*
808 1. Include necessary headers for shared memory operations and standard I/O.
809 2. Define a unique key for shared memory creation.
810 3. Create a shared memory segment using shmget() with the defined key and required size.
811     - If creation fails, print an error message and exit.
812 4. Attach the shared memory segment to the process's address space using shmat().
813     - If attachment fails, print an error message and exit.
814 5. Write the process ID of the current process into the shared memory.
815 6. Print a message indicating that the process ID has been written to shared memory.
816 7. Detach the shared memory segment from the process's address space using shmdt().
817 8. Exit the program.
818
819 Summary:
820 This program (program1.c) demonstrates the creation and usage of shared memory in a Linux
821 environment. It writes the process ID of the current process into a shared memory segment, which
822 can be read by another program (program2.c). The shared memory segment is properly detached after
823 use.
824 */
825 #include <stdio.h>
826 #include <sys/ipc.h>

```

```

825 #include <sys/shm.h>
826 #include <unistd.h>
827
828 int main() {
829     key_t key = 1234; // Unique key for shared memory
830     int shmid = shmget(key, sizeof(int), 0666 | IPC_CREAT); // Create shared memory
831     if (shmid == -1) {
832         perror("shmget failed");
833         return 1;
834     }
835
836     int *shared_data = (int *)shmat(shmid, NULL, 0); // Attach memory
837     if (shared_data == (int *)-1) {
838         perror("shmat failed");
839         return 1;
840     }
841
842     *shared_data = getpid(); // Write Process ID to shared memory
843     printf("Process 1 (PID: %d) wrote to shared memory.\n", *shared_data);
844
845     shmdt(shared_data); // Detach memory
846     return 0;
847 }
848
849 // Assignment 6 -a (program2.c)
850 /*
851 * 1. Include necessary headers for shared memory operations and standard I/O.
852 * 2. Define a unique key for shared memory identification.
853 * 3. Access an existing shared memory segment using shmget with the defined key.
854 *      - If shmget fails, print an error message and exit.
855 * 4. Attach the shared memory segment to the process's address space using shmat.
856 *      - If shmat fails, print an error message and exit.
857 * 5. Read the integer value from the shared memory and print it.
858 * 6. Detach the shared memory segment from the process using shmdt.
859 * 7. Remove the shared memory segment using shmctl with IPC_RMID command.
860 * 8. Exit the program.
861 */
862
863 /*
864 * Summary:
865 * This program (program2.c) demonstrates shared memory usage by reading data written by another
866 * process (program1.c).
867 * It accesses an existing shared memory segment, reads the data, and then cleans up by detaching
868 * and removing the shared memory.
869 */
870 #include <stdio.h>
871 #include <sys/ipc.h>
872 #include <sys/shm.h>
873
874 int main() {
875     key_t key = 1234;
876     int shmid = shmget(key, sizeof(int), 0666); // Access existing shared memory
877     if (shmid == -1) {
878         perror("shmget failed");
879         return 1;
880     }

```

```

879
880     int *shared_data = (int *)shmat(shmid, NULL, 0); // Attach memory
881     if (shared_data == (int *)-1) {
882         perror("shmat failed");
883         return 1;
884     }
885
886     printf("Process 2 read from shared memory: %d\n", *shared_data);
887
888     shmdt(shared_data); // Detach memory
889     shmctl(shmid, IPC_RMID, NULL); // Remove shared memory
890     return 0;
891 }
892
893 // Assignment 6 -b (process_time.c)
894 /*
895 * 1. Include necessary headers for file operations, process handling, and time measurement.
896 * 2. Define the main function.
897 * 3. Declare variables to store start and end times using `struct timeval`.
898 * 4. Open a file named "process_time.txt" in write mode to store the process times.
899 * 5. Record the submission time using `gettimeofday()` and store it in the `start` variable.
900 * 6. Create a child process using `fork()` .
901 * 7. In the child process:
902 *     a. Print a message indicating the child process is running.
903 *     b. Simulate some execution using `sleep()` .
904 *     c. Exit the child process.
905 * 8. In the parent process:
906 *     a. Wait for the child process to finish using `wait()` .
907 *     b. Record the finish time using `gettimeofday()` and store it in the `end` variable.
908 *     c. Write the submission and finish times to the file in seconds and microseconds.
909 *     d. Close the file.
910 *     e. Print a message indicating the times have been recorded.
911 * 9. Return 0 to indicate successful execution.
912 *
913 * Summary:
914 * The program creates a child process and measures its submission and finish times using the
`gettimeofday()` system call. These times are written to a file named "process_time.txt". The
`fork()` system call is used to create the child process, and `wait()` ensures the parent waits for
the child to complete. The program demonstrates basic process handling and time measurement in C.
915 */
916 #include <stdio.h>
917 #include <sys/types.h>
918 #include <sys/wait.h>
919 #include <unistd.h>
920 #include <sys/time.h>
921 #include <stdlib.h>
922
923 int main() {
924     struct timeval start, end;
925     FILE *file = fopen("process_time.txt", "w"); // Open file to store times
926
927     gettimeofday(&start, NULL); // Record submission time
928     pid_t pid = fork(); // Create child process
929
930     if (pid == 0) { // Child process
931         printf("Child process is running...\n");

```

```

932     sleep(3); // Simulate execution
933     exit(0);
934 } else { // Parent process
935     wait(NULL); // Wait for child to finish
936     gettimeofday(&end, NULL); // Record finish time
937
938     fprintf(file, "Submission Time: %ld.%06ld seconds\n", start.tv_sec, start.tv_usec);
939     fprintf(file, "Finish Time: %ld.%06ld seconds\n", end.tv_sec, end.tv_usec);
940     fclose(file);
941
942     printf("Process times recorded in 'process_time.txt'\n");
943 }
944 return 0;
945 }
946
947
948 // Assignment 7 a (program1.c)
949 /*
950 1. Include necessary headers for semaphore operations and file handling.
951 2. Define a function `semaphore_signal` to increment the semaphore value by 1.
952 3. In the `main` function:
953     a. Generate a unique key for the semaphore using `key_t`.
954     b. Create a semaphore set with one semaphore using `semget`.
955     c. Initialize the semaphore value to 0 using `semctl`.
956     d. Open the file "sample.txt" in read mode.
957     e. Read the contents of the file character by character and print them to the console.
958     f. Close the file after reading.
959     g. Signal the semaphore to indicate that Program1 has completed reading the file.
960     h. Print a message indicating that Program1 has completed and signaled Program2.
961 4. End the program.
962
963 Summary:
964 This program (Program1) reads the contents of a file and uses a semaphore to synchronize with
another program (Program2). The semaphore ensures that Program2 can only read the file after
Program1 has completed its reading. The semaphore is initialized to 0 and incremented by 1 after
Program1 finishes reading, signaling Program2 to proceed.
965 */
966 #include <stdio.h>
967 #include <sys/ipc.h>
968 #include <sys/sem.h>
969 #include <stdlib.h>
970
971 void semaphore_signal(int semid) {
972     struct sembuf sb = {0, 1, 0}; // Increment semaphore by 1
973     semop(semid, &sb, 1);
974 }
975
976 int main() {
977     key_t key = 1234;
978     int semid = semget(key, 1, 0666 | IPC_CREAT); // Create semaphore
979
980     // Initialize semaphore to 0 initially
981     semctl(semid, 0, SETVAL, 0);
982
983     FILE *file = fopen("sample.txt", "r");
984

```

```

985     char ch;
986     printf("Program1 is reading the file:\n");
987     while ((ch = fgetc(file)) != EOF) {
988         printf("%c", ch);
989     }
990     fclose(file);
991
992     // After reading, signal program2
993     semaphore_signal(semid);
994     printf("\nProgram1 completed. Signaled Program2.\n");
995
996     return 0;
997 }
998 */
999 /*
1000 * 1. Include necessary headers for semaphore and file operations.
1001 * 2. Define a function `semaphore_wait` to decrement the semaphore value (wait operation).
1002 * 3. In the `main` function:
1003 *     a. Generate a unique key for the semaphore using `key_t` .
1004 *     b. Access the existing semaphore set using `semget` .
1005 *     c. Wait for the semaphore to be released by program1 using `semaphore_wait` .
1006 *     d. Open the file "sample.txt" in read mode.
1007 *     e. Read and print the contents of the file character by character.
1008 *     f. Close the file after reading.
1009 *     g. Remove the semaphore using `semctl` to clean up resources.
1010 *     h. Print a message indicating program2 has completed.
1011 */
1012
1013 /*
1014 * Summary:
1015 * This program (program2.c) ensures synchronization with program1.c using semaphores.
1016 * It waits for program1 to finish reading the file before it starts reading the same file.
1017 * After reading, it removes the semaphore to clean up resources.
1018 */
1019 // Assignment 7a - program2
1020 #include <stdio.h>
1021 #include <sys/ipc.h>
1022 #include <sys/sem.h>
1023 #include <stdlib.h>
1024
1025 void semaphore_wait(int semid) {
1026     struct sembuf sb = {0, -1, 0}; // Decrement semaphore by 1
1027     semop(semid, &sb, 1);
1028 }
1029
1030 int main() {
1031     key_t key = 1234;
1032     int semid = semget(key, 1, 0666); // Access existing semaphore
1033
1034     // Wait until program1 completes
1035     semaphore_wait(semid);
1036
1037     FILE *file = fopen("sample.txt", "r");
1038
1039     char ch;
1040     printf("Program2 is reading the file after Program1:\n");

```

```

1041     while ((ch = fgetc(file)) != EOF) {
1042         printf("%c", ch);
1043     }
1044     fclose(file);
1045
1046     // Remove the semaphore after use
1047     semctl(semid, 0, IPC_RMID);
1048     printf("\nProgram2 completed and semaphore removed.\n");
1049
1050     return 0;
1051 }
1052
1053 // Assignment 7 b (program1.c)
1054 /*
1055 * 1. Include necessary headers for shared memory and semaphore operations.
1056 * 2. Define a function to signal (increment) the semaphore.
1057 * 3. In the main function:
1058 *     a. Generate a unique key for shared memory and semaphore.
1059 *     b. Create a shared memory segment and a semaphore using the key.
1060 *     c. Attach the shared memory segment to the process's address space.
1061 *     d. Write a string (e.g., "Hello from Shared Memory!") to the shared memory.
1062 *     e. Initialize the semaphore to 0 to indicate it's locked.
1063 *     f. Print a message indicating data has been written to shared memory.
1064 *     g. Signal the semaphore to notify the other process (prog2) that data is ready.
1065 *     h. Detach the shared memory segment from the process's address space.
1066 * 4. Exit the program.
1067
1068 * Summary:
1069 * prog1.c writes a string to shared memory and uses a semaphore to signal prog2.c
1070 * that the data is ready. It demonstrates inter-process communication using shared
1071 * memory and semaphores.
1072 */
1073 // Assignment 7b - program1
1074 #include <stdio.h>
1075 #include <sys/ipc.h>
1076 #include <sys/shm.h>
1077 #include <sys/sem.h>
1078 #include <stdlib.h>
1079
1080 void semaphore_signal(int semid) {
1081     struct sembuf sb = {0, 1, 0}; // Increment semaphore
1082     semop(semid, &sb, 1);
1083 }
1084
1085 int main() {
1086     key_t key = 5678;
1087     int shmid = shmget(key, 100, 0666 | IPC_CREAT);
1088     int semid = semget(key, 1, 0666 | IPC_CREAT);
1089
1090     char *str = (char *)shmat(shmid, NULL, 0);
1091     sprintf(str, "Hello from Shared Memory!");
1092
1093     semctl(semid, 0, SETVAL, 0); // Initialize semaphore to 0
1094
1095     printf("prog1: Data written to shared memory.\n");
1096 }
```

```

1097     semaphore_signal(semid); // Signal prog2 to read
1098     shmdt(str); // Detach shared memory
1099
1100     return 0;
1101 }
1102 /*
1103 1. Include necessary headers for shared memory and semaphore operations.
1104 2. Define a function `semaphore_wait` to decrement the semaphore value (wait operation).
1105 3. In the `main` function:
1106     a. Generate a unique key for shared memory and semaphore.
1107     b. Access the shared memory segment and semaphore created by prog1.
1108     c. Wait for prog1 to write data into the shared memory using `semaphore_wait`.
1109     d. Attach the shared memory segment to the process's address space.
1110     e. Read and print the data from the shared memory.
1111     f. Detach the shared memory segment from the process's address space.
1112     g. Remove the semaphore and shared memory segment using appropriate system calls.
1113     h. Print a message indicating successful cleanup.
1114 4. End the program.
1115
1116
1117 Summary:
1118 This program (prog2.c) reads data from a shared memory segment written by prog1.c.
1119 It uses a semaphore to synchronize access, ensuring prog1 writes data before prog2 reads it.
1120 After reading, it cleans up by removing the semaphore and shared memory.
1121 */
1122 // Assignment 7b - program2
1123 #include <stdio.h>
1124 #include <sys/ipc.h>
1125 #include <sys/shm.h>
1126 #include <sys/sem.h>
1127 #include <stdlib.h>
1128
1129 void semaphore_wait(int semid) {
1130     struct sembuf sb = {0, -1, 0}; // Decrement semaphore
1131     semop(semid, &sb, 1);
1132 }
1133
1134 int main() {
1135     key_t key = 5678;
1136     int shmid = shmget(key, 100, 0666);
1137     int semid = semget(key, 1, 0666);
1138
1139     semaphore_wait(semid); // Wait for prog1 to write
1140
1141     char *str = (char *)shmat(shmid, NULL, 0);
1142     printf("prog2: Data read from shared memory: %s\n", str);
1143     shmdt(str); // Detach shared memory
1144
1145     // Remove semaphore and shared memory
1146     semctl(semid, 0, IPC_RMID);
1147     shmctl(shmid, IPC_RMID, NULL);
1148     printf("prog2: Semaphore and shared memory removed.\n");
1149
1150     return 0;
1151 }

```