02-06-2018, 02-22-2018

Process Creation and Handling Problems reviewed during lecture time.

Problem 1

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

   pid = fork();

   if (pid == 0) { /* child process */
      value += 15;
      return 0;
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE A */
      return 0;
   }
}
```

Figure 3.30 What output will be at Line A?

Solution: The result is still 5 as the child updates its copy of value. When control returns to the parent, its value remains at 5.

Problem 2

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* fork a child process */
    fork();

    /* fork another child process */
    fork();

    /* and fork another */
    fork();

    return 0;
}
```

Figure 3.31 How many processes are created?

Solution: There are 8 processes created. Can you see why?

Problem 3

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    for (i = 0; i < 4; i++)
      fork();

    return 0;
}
```

Figure 3.32 How many processes are created?

Solution: 8 processes are created. Why? Try this …

Problem 4

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
       printf("LINE J");
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.33 When will LINE J be reached?

Solution: The call to *exec*() replaces the address space of the process with the program specified as the parameter to exec(). If the call to *exec*() succeeds, the new program is now running and control from the call to exec() never returns. In this scenario, the line *printf*("Line J"); would never be performed. However, if an error occurs in the call to *exec*(), the function returns control and therefor the line printf("Line J"); would be performed.

Problem 5

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
}
```

Figure 3.34 What are the pid values?

Solution: A = 0, B = 2603, C = 2603, D = 2600

Problem 6

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
int i;
pid_t pid;

   pid = fork();

   if (pid == 0) {
      for (i = 0; i < SIZE; i++) {
         nums[i] *= -i;
         printf("CHILD: %d ",nums[i]); /* LINE X */
      }
   }
   else if (pid > 0) {
      wait(NULL);
      for (i = 0; i < SIZE; i++)
         printf("PARENT: %d ",nums[i]); /* LINE Y */
   }

   return 0;
}
```

Figure 3.35 What output will be at Line X and Line Y?

Solution: Because the child is a copy of the parent, any changes the child makes will occur in its copy of the data and won't be reflected in the parent. As a result, the values output by the child at line X are 0, -1, -4, -9, -16. The values output by the parent at line Y are 0, 1, 2, 3, 4

Problem 7: In the program of the following page you are experimenting with a LINUX shell where the **first existing shell** process starts from number 4167 and this number increases by one for every additional process that is created/forked.

Defs: Sys Calls: `read, getpid, getppid, exec, exit, waitpid`. Sys call `waitpid()` blocks on an existing child of the calling process:

A. **(4 marks)** What is printed at the output with this code? Please show/justify with details.

B. **(6 marks)** Can you modify (remove or move) one or more existing line(s) in this code in order to ensure all the intended printouts are executed? Which and how? List all the possible printouts for your solution (if more than one) and justify your answer(s). Omit the while loop for this question.

C. **(6 marks)** In addition to your answer to part B, can you select and add **only one new system call** that modifies and uniquely determines the relationship of the processes generated? Please show where you add this line in the existing code. Explain how and why your printout is different now. You may omit the while loop for answering this question.

D. **(7 marks)** Are there any circumstances under which it's still possible that the printouts you obtain for your answer in part B are the same as the printouts that you obtain for your answer in part C? If not why, if so under which system call and with which arguments? Omit the while loop here too.

E. **(7 marks)** Let's go back to part A and just remove the first exec sys call encountered while all the rest remains the same. Assume you start this shell program, and type "ls" once. What will happen?

F. **(6 marks)** With this shell implementation you just described in part E, what is the impact, if any, on a long running OS?

```
shell (..) {
while (1) {
pid_t pid;
char *args = read_args();
int status = -1;
int value = 8;
.. ..
exec (ls, args);
value++;
pid = fork();
if (pid== -1){
            perror ("fork failure");
            exit (EXIT_FAILURE);
}
else if (pid) {
value++;
exec (ls, args);
printf ("I am process %d and I report to process %d value=%d\n", pid,getppid(),value);
}
else {
    waitpid(pid);
    printf ("I am process %d and I report to process %d value=%d\n", getpid(),
getppid(),value);
}
}                              // end while loop
}                              // end program
```

Solution:

A. **(4 marks)** Shell's address space is entirely replaced with the new command (*ls*), therefore the shell will terminate once *ls* is terminated. Printed output is what is contained in the command *ls* (…, *args*): where *args* can be the path to a folder whose files will be listed. Nothing else is printed as output, as sys call *exec* executes the command included in its arguments and exits.

What about the *waitpid(pid)* sys call? In essence the child process has invoked sys call: *waitpid(*0*)*. But does the child have a child with valid *pid*==0? No, the child does not have any children, but even if it did, their *pids* would be at least >1. Hence sys call *waitpid* does not have an effect on the child. It returns immediately. Hence, we do not need to remove this sys call.

B. **(6 marks)** There can be many solutions to this part.
It's straightforward to remove sys calls *exec (ls, , …, args);* completely or place them in the jurisdiction of the parent and/or child processes but below the *printf* commands. It may be the case though that processes exit before the printouts from the *printf* appear on the screen (depending on our OS version).

Order of printouts: The two processes, parent and child run concurrently on the processor. The process which occupies the processor for a number of clock cycles (time units) every time, is determined by the scheduling policy of the OS.

So, based on the above, the parent and child printouts in **random order** are the following:

Parent:
```
I am process 4168 and I report to process 1 value=10
```

Child:
```
I am process 4168 and I report to process 4167 value=9
```

OR

Child:
```
I am process 4168 and I report to process 4167 value=9
```

Parent:
```
I am process 4168 and I report to process 1 value=10
```

C. **(6 marks)**
There's already a relationship between parent-child processes as the parent forks the child. However, considering the way the above code is written, one observes that after all both

processes act completely independent, without any point of synchronization or data transfer that could define another deterministic relationship between the two.
One simple way to reshape the relationship of the two processes and make it deterministic without changing the above code too much, is to insert synchronization in the printouts of both processes. This is to say, make the printout of this program absolutely predictable, either have the parent print first and the child second or vice versa.

Solution 1: include in the parent's code sys call *waitpid*(*pid*) or *wait()* before *printf*(). Then, the parent blocks waiting for the child process to finish execution and exit. The proper signal of child termination will be received by the parent and *waitpid*(*pid*) will unblock. Then the parent is released and proceeds to execute *printf*. The sys call *wait ()* blocks the calling process until one of its *child* processes exits/completes or a signal is received. In our case we only have one child process anyway. If more than one children is running, then *wait()* returns the first time one of the children exits.

Solution 2: the parent invokes sys call *sleep (arg)*. If we fine-tune the number of seconds in the argument, e.g., *sleep (500),* we give the other process the opportunity to issue its printout first. If you increase *arg* to a very big constant then it is almost 100% that the other process prints first.

D. **(7 marks)** At any moment only one of the two processes occupies the processor for a number of time units in a time multiplexing fashion:

i) Assume that every process occupies the processor for *X* time units and then yields to the other process. Assume further that you selected in Part C to insert *sleep* (*Y*) above the printouts of either the parent or the child.
Without loss of generality, further assume that in Part B the parent got to execute first. So, if we place *sleep()* before the printouts of the child, we provide the parent even more time (opportunities) to execute *printf* first. A child that has called *sleep(arg)* either occupies the processor by remaining idle and having its timer decreasing, or can even be blocked to run and move to the blocked process queue until the "*sleep*" timer expires, giving the parent exclusive use of the processor. If however, we place sys call *sleep (Y)* before the printouts of the parent, the parent can still be the one who occupies the processor if, roughly speaking, $X > Y$. In this case, our answers for part B and C can be the same. In the opposite case, the sleep will cause a reversal in the printout output.

ii) However, a sys call *wait ()* on behalf of the parent above its printouts, ensures that its valid child will always print first. If in Part B the parent printed first, then now Part B and Part C cannot be equivalent under any circumstances.

E. **(7 marks)** The parent (shell) executes *ls*, terminates/exits via sys call *exec* and goes into zombie state. The child calls *waitpid(0)*, and returns immediately since 0 is not valid PID. It then goes back into the loop as it's the only live process now and becomes a shell.

F. **(6 marks)** The above process described in Part E repeats on every iteration of the while loop. There will be a lot of zombie processes and the OS will eventually run out of *pids* and resources. Since the parent will die before the child, the latter will be adopted by *init*, which will wait the child to terminate and then reap it.