Lecture 11. Process Control

Race Conditions

A **race condition** occurs when multiple processes access shared data, and the outcome depends on the order in which the processes run. The `fork()` function is likely to generate race conditions if the logic after the fork depends on the order in which the parent or the child runs first. The order of execution of these processes depends on the system load and on the kernel's scheduling algorithm.

We saw that, if a process wants to wait for its child to terminate, it must call one of the `wait()` functions. If a process wants to wait for its parent to terminate, it could use the following loop:

```
while (getppid()!=1)
     sleep(1);
```

This says, within the branch executing the child, put in a loop waiting for the child to become inherited by `init` (whose process ID is 1), thus guaranteeing that the parent has terminated.

The following shows the example from Lecture 11 modified to include this loop:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */

#define BUFFSIZE 4096
#define FILESIZE 1474560

void Input(int, char *);

int main(int argc, char *argv[])
{
     int fd=open("/Users/awise/Stevens/Lecture11/file1.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);
     int pid=fork();
     char *process;
     if (pid==0) /* child process */
     {
          process="Child process:\n";
          Input(fd, process);
          while (getppid()!=1) /* while parent's parent is not
init */
               sleep(2);
     }
     else /* parent process */
     {
          process="Parent process:\n";
          Input(fd, process);
     }
return 0;
}

void Input(int filedes, char *which)
{
     write(filedes, which, strlen(which));
     char **name=calloc(3, 20*sizeof(char));
     char **it;
     name[0]="Adriana Wise\n";
     name[1]="Richard Stevens\n";
     name[2]="Evi Nemeth\n";
     for (it=&name[0]; *it!=NULL; it++)
     {
          ssize_t num_bytes=write(filedes, *it, strlen(*it));
     }
}
```

The text file will be written sequentially, with the parent doing the work and terminating first:

```
Parent process:
Adriana Wise
Richard Stevens
Evi Nemeth
Child process:
Adriana Wise
Richard Stevens
Evi Nemeth
```

This type of loop, called **polling**, wastes CPU time, since the caller (the child) is woken up every 2 seconds to test the condition (whether the parent ID of the child has become 1, thus the child was orphaned and inherited by `init`).

The following program, from APUE (fixed!), similar in concept to our version of the file sharing program above which produced an interleaved output in the text file, uses `putc()` to output a string, character at a time, at the command line. Because of the `sleep(1)` call in the `STDOutput()` function, the kernel is forced to switch between the parent and the child every character, and the output is also interleaved, thus illustrating a **race condition**:

```
#include <sys/types.h> /* for fork() */
#include <unistd.h> /* for fork() */
#include <stdio.h> /* for putc() */
#include <string.h> /*for strlen() */

static void STDOutput(char *string);

int main(void)
{
     pid_t pid;
     pid=fork();
     if (pid==0) /* child process */
     {
          STDOutput("Output from the child.\n");
     }
     else
     {
          STDOutput("Output from the parent.\n");
     }
return 0;
}

static void STDOutput(char *string)
{
     char *ptr;
     int c;
     int n=strlen(string);

     setbuf(stdout, NULL);
     for (ptr=&string[0]; ptr!=&string[n]; ptr++)
     {
          c=*ptr;
          putc(c, stdout);
          sleep(1);
     }
}
```

The output is:

```
$ ./myrace
OOuuttppuutt  ffrroomm  tthhee  cpharielnd.t
.
```

To prevent race conditions and to avoid polling, signaling and interprocess communication is required between multiple processes. These will be studied later.

The `exec()` Functions

One use of the `fork()` function is to create a new process (the child) which then causes another program to execute, by calling one of the `exec()` functions. When a process calls an `exec()` function, that process is completely replaced by the new program. The new program starts executing at its `main()` function.

The process ID does not change at the call of `exec()`, because the call of `exec()` does not produce a *new* process. `exec()` only replaces the current process (text, data, heap, stack) with a new program loaded from disk.

There are 6 `exec()` functions. Again, `fork()` creates new processes, while `exec()` initiates new programs. The `exit()` function and the `wait()` functions handle termination and waiting for termination.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, …, (char *)0);

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, …, (char *)0,
          char *const envp[]);

int execve(const char *pathname, char *const argv[], char *const
          envp[]);

int execlp(const char *filename, const char *arg0, …, (char*)0);

int execvp(const char *filename, char *const argv[]);
```

All 6 return -1 on error, no return on success.

Arguments:

- *`pathname`*—a full directory path to an executable file
- *`filename`*—if absolute path, same as pathname; if relative path, it is relative to the value(s) of the `PATH` environment variable (e.g. `PATH=/bin:/usr/bin:/usr/local/bin:.`)
- *`arg0, …`*—pointers to command line arguments of the new program
- `(char *) 0`—null terminating character for command line args passed one by one
- *`argv[]`*—pointer to an array of pointers to command line args of new program

If file is not a machine executable file generated by the link editor, `execlp()` and `execvp()` assume file is a shell script and invoke `/bin/sh` with *`filename`* as input to the shell (command=file).

The suffixes `l` and `v` mean list and vector, and refer to the way of passing the argument lists: the `l` functions take each of the command line arguments to the new program as separate arguments; the `v` functions take these as the address to an array of pointers previously built with these arguments.

The suffix `e` means environment, and refers to the way of passing the environment strings: the `e` functions take a pointer to an array of pointers to the environment strings; all other functions use the `environ` variable of the calling process to copy this environment into (for) the new program.

The suffix `p` means pathname, and refers to the way of passing the filename: the `p` functions take a relative path, and use the PATH environment variable to do the search for the executable; all other functions take an absolute pathname.

Function	**pathname**	**filename**	**Arg list**	**argv[]**	**`environ`**	**envp[]**
`execl`	•		•		•	
`execlp`		•	•		•	
`execle`	•		•			•
`execv`	•			•	•	
`execvp`		•		•	•	
`execve`	•			•		•
(letter in name)		`p`	`l`	`v`		`e`

The previous table shows the arguments to these 6 functions.

The following example program calls `execv()` to cause the child process to call another program, `myprog.c`, to execute with a list of command line arguments, and to cause the parent process to call the same program, `myprog.c`, with a different list of command line arguments. Both processes share the same output file, `file2.txt`. Below is `myprog.c`:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */
#include <stdarg.h> /* for va_start(), va_arg(), va_end() */

#define BUFFSIZE 4096
#define FILESIZE 1474560

void Input(int, int, ...);

int main(int argc, char *argv[])
{
     int fd=open("/Users/awise/Stevens/Lecture11/file2.txt",
O_CREAT|O_RDWR|O_APPEND, S_IRUSR|S_IWUSR);

/*
     char **iterator;
     for (iterator=&argv[0]; *iterator!=NULL; iterator++)
     {
          printf("%s\n", *iterator);
     }
*/
     Input(fd, argc-1, argv[1], argv[2], argv[3]);
return 0;
}

void Input(int filedes, int num, ...)
{
     va_list arguments;
     ssize_t num_bytes;

     va_start(arguments, num);
     char **names=calloc(3, 20*sizeof(char));
     char *name=malloc(20*sizeof(char));
     for (int x=0; x<num; x++)
     {
          name=va_arg(arguments, char *);
          printf("%s\n", name);
          names[x]=name;
          num_bytes=write(filedes, names[x], strlen(name));
          num_bytes=write(filedes, "\n", strlen("\n"));
     }
     va_end(arguments);
}
```

The caller program calls `fork()` and `execv()` with the above program as an *executable file* as argument:

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */

#define BUFFSIZE 4096
#define FILESIZE 1474560
#define LISTSIZE 20

int main(int argc, char *argv[])
{
     char *pathname="/Users/awise/Stevens/Lecture11/myprog";
     //char **const list1=malloc(LISTSIZE*sizeof(char *));
     char *list1[LISTSIZE];
     list1[0]=pathname;
     list1[1]="Adriana Wise";
     list1[2]="Richard Stevens";
     list1[3]="Evi Nemeth";
     list1[4]=NULL;

     //char **const list2=malloc(LISTSIZE*sizeof(char *));
     char *list2[LISTSIZE];
     list2[0]=pathname;
     list2[1]="Alicia Beth Moore";
     list2[2]="Tom Petty";
     list2[3]="Sam Smith";
     list2[4]=NULL;

     int pid=fork();
     if (pid==0) /* child process */
     {
          execv(pathname, list1);
          while (getppid()!=1) /* while parent's parent is not 
init */
               sleep(2);
     }
     else /* parent process */
     {
          execv(pathname, list2);
     }
return 0;
}
```

Changing User IDs and Group IDs

When our programs need additional privileges or to gain access to resources that they aren't currently allowed to access, they need to change their user or group IDs to an ID with appropriate permissions. The same thing goes in the other direction, in order to prevent programs to have access to certain resources.

A good design practice is to only endow programs with the minimum set of privileges that still allows them to do the job. This reduces the likelihood of a malicious attack using weaknesses of a program with access to more resources than it is entitled to, relative to its job.

The real user ID and effective user ID can be set with the `setuid()` function. Similarly, the real group ID and the effective group ID can be set with the `setgid()` function:

```
#include <unistd.h>

int setuid(uid_t uid);

int setgid(uid_t gid);
```

Both return 0 if OK, -1 if error.

The rules on who can change the user ID are set below. The same principles apply to the group ID:

1. If the process has root (superuser) privileges, `setuid()` sets the real ID, the effective ID, and the saved set-user-ID to *`uid`*.

2. If the process doesn't have root privileges, but *`uid`* equals either the real user ID or the saved set-user-ID, `setuid()` sets only the effective user ID to *`uid`* (i.o.w. only the minimal settings are allowed).
3. If *`uid`* equals neither the real user ID, nor the saved set-user-ID, `errno` is set to EPERM, and `setuid()` returns -1.

The following table summarizes how the 3 user IDs can be changed:

ID	**`exec`**		**`setuid(uid)`**	
	set-user-ID bit off	**set-user-ID bit on**	**superuser**	**unprivileged user**
real user ID	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-user ID	copied from effective user ID	copied from effective user ID	set to uid	unchanged

The following functions, available for BSD, allow the swapping of the real user ID and the effective user ID (re—real):

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);
```

Both return 0 if OK, -1 if error.

Finally, the functions `seteuid()` and `setegid()` allow only the changing of the effective user ID or the effective group ID:

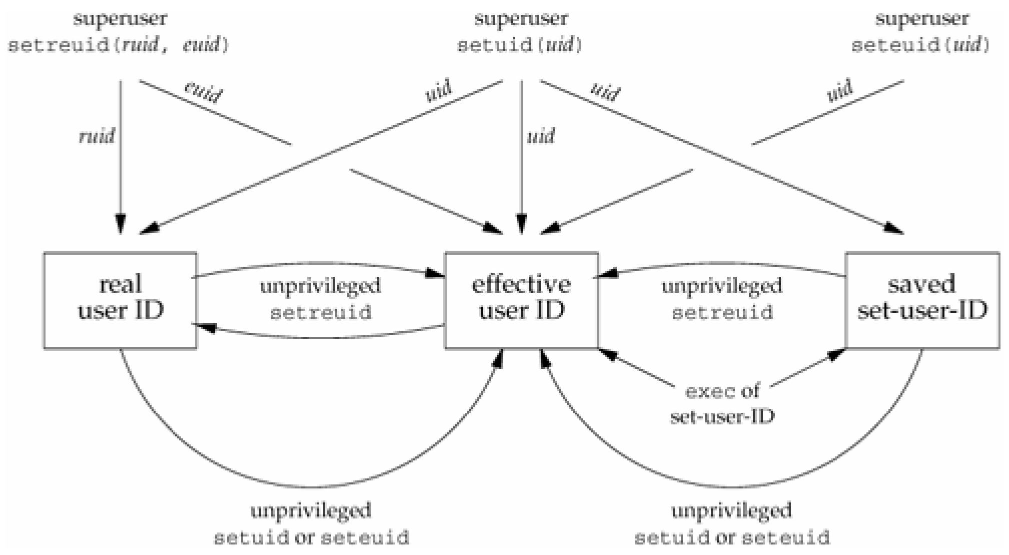
```
#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);
```

Both return 0 if OK, -1 if error.

The relationship between the 3 user IDs and the setter functions is shown in the following diagram. Group IDs follow a similar principle.



Interpreter Files

A shell script, for instance, is an interpreter file. The following script calculates (in floating point arithmetic, no less!) square standard deviations of 4 floating point numbers:

```
#!/bin/sh

#take 4 arguments from command line
#stuff them in an array
argc=1
declare -a array
while [[ $argc -le $# ]]
do
        array[$argc-1]=`echo ${!argc}`
        argc=$(( $argc+1 )) #same as argc=`expr $argc+1`
done
#calculate sum
sum=0
i=0
while [[ $i -lt ${#array[*]} ]]
do
        sum=$(echo "$sum+${array[$i]}" | bc)
        i=$(( i+1 ))
done
echo $sum
echo ${#array[*]}
```

```
#calculate average
average=$(echo "`expr $sum/${#array[*]}`" | bc -l)
printf "%.2f\n" $average
#calculate standard deviation
i=0
declare -a stddev
while [[ $i -lt ${#array[*]} ]]
do
        #calculate (s_i-a)^2
        #temp=$(echo "${array[$i]}-$average" | bc)
        stddev[$i]=$(echo "(${array[$i]}-$average)^2" | bc)
        printf "%.2f\n" ${stddev[$i]}
        i=$(( i+1 ))
done
```

The following adapted C program will execute this shell script by calling `execv()` on the pathname of this script, `/Users/awise/UNIX1/Lecture12/statistics.v.5`, with the 4 numbers the script would normally take as command line arguments, passed in to `execv()` as a `char *` array. The numbers are passed in as strings, and the script correctly deals with them as numbers.

```
#include <unistd.h> /* for read(), write() */
#include <fcntl.h> /* for open() */
#include <string.h> /* for strlen() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for calloc() */

#define BUFFSIZE 4096
#define FILESIZE 1474560
#define LISTSIZE 20

int main(int argc, char *argv[])
{
     char *pathname="/Users/awise/UNIX1/Lecture12/statistics.v.
5";
     char **const list1=malloc(LISTSIZE*sizeof(char *));
     //float *const list1=malloc(LISTSIZE*sizeof(float));
     list1[0]=pathname;
     list1[1]="1.1";
     list1[2]="2.1";
     list1[3]="3.1";
     list1[4]="4.1";
     list1[5]=NULL;

     char **const list2=malloc(LISTSIZE*sizeof(char *));
     //float *const list2=malloc(LISTSIZE*sizeof(float));
     list2[0]=pathname;
     list2[1]="10";
     list2[2]="20";
     list2[3]="30";
     list2[4]="40";
     list2[5]=NULL;

     int pid=fork();
     if (pid==0) /* child process */
     {
          execv(pathname, list1);
          while (getppid()!=1) /* while parent's parent is not
init */
               sleep(2);
     }
     else /* parent process */
     {
          execv(pathname, list2);
     }
return 0;
}
```

This is the original output from the script itself:

```
$ ./statistics.v.5 1.1 2.1 3.1 4.1
10.4
4
2.60
2.25
0.25
0.25
2.25
```

This is the output from the calling program (which, remember, has a child and a parent. The numbers below represent *both outputs*):

```
$ ./myexec3
100
4
10.4
4
2.60
25.00
2.25
```

```
225.00
25.00
0.25
0.25
25.00
225.00
2.25
```

The `system()` Function

This function executes a command string from within a program. Its synopsis is:

```
#include <stdlib.h>

int system(const char *cmdstring);
```

Returns: see below.

Because `system()` is implemented calling `fork()`, `exec()`, and `waitpid()`, there are 3 types of return values:

1. If either `fork()` fails or `waitpid()` returns an error other than EINTR, `system()` returns -1 with `errno` set appropriately.
2. If `exec()` fails, the return value is as if the shell had executed `exit(127)`.
3. If all 3 succeed, `system()` returns the termination status from the shell.

Below is an example of `system()` call (sic) that calls "`date`":

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
     int status;

     status=system("date");
     printf("Exit status=%d\n", status);
return 0;
}
```

This is its output:

```
$ ./mysystem
Tue Mar 10 18:49:44 EDT 2015
Exit status=0
```

We can also call a shell script, `/Users/awise/UNIX1/Lecture12/test4` (this was the stage where I challenged myself to do floating point arithmetic in the shell against all odds):

```
#!/bin/sh

result=`{
        sum=$( echo 3.2+3.5 | bc)
        echo $sum
        echo $sum*10
        var=$(echo "$sum*10" | bc)
        echo $var
        }`
echo “Result=$result"
```

The caller program, using `system()`, would be:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
     int status;
     char *pathname;

     pathname="/Users/awise/UNIX1/Lecture12/test4";
     status=system(pathname);
     printf("Exit status=%d\n", status);
return 0;
}
```

The output from the caller program is:

```
$ ./mysystem2
Result=6.7
6.7*10
67.0
Exit status=0
```