CS 3013 - Operating Systems

Project 1 (80 points)
Assigned: Friday, January 29, 2021
Checkpoint: Friday, February 5, 2021
Due: Tuesday, February 9, 2021

# Project 1: The Process Party

Processes have a strange existence in the Linux or Unix world. The process creation, termination, and execution model may take experience to understand. In this project, we will take a practice-centric approach to exploring how processes work, how they share information, and how they are organized.

Our exploration will involve five short scenarios, each of which explores a different concept. An overview of the five scenarios is as follows:

1. **The Prolific Parent:** A parent process spawns a random number of children (between 10 and 15). Each child introduces itself, waits for a short time, and then terminates (with a specifically chosen exit code). This scenario's code should be in `prolific.c` and the `Makefile` should compile it to an executable named `prolific`.

2. **Lifespan Generation:** A parent process picks a random number (between 7 and 11) for the number of lifespans that will be explored. It then creates a child. That child decrements the lifespan count, and, if the count is non-zero, creates its own child (which repeats the process) and prints out a statement indicating its process ID and that of its child. If a child decrements the count and it becomes zero (or somehow becomes negative), the child exits with an exit code of 0. All processes wait until their child returns and exits with a value that is one greater than its child's exit value. When the initial parent process confirms its child has exited, it prints out the exit value of its child and itself exits. This scenario's code should be in `generation.c` and the `Makefile` should compile it to an executable named `generation`.

3. **The Explorer:** A parent process moves around the file system five times, randomly selecting a directory each time (from among `/home`, `/proc`, `/proc/sys`, `/usr`, `/usr/bin`, and `/bin`). Each time it changes its current working directory (e.g., using `chdir()`), it indicates which string it chose and outputs the value of the current working directory (using `getcwd()`). It then forks off a child that runs the `ls -alh` command. Done correctly, the child's listing output should match the directory the parent selected. The parent waits on the child to complete before moving on to the next file system location. This scenario's code should be in `explorer.c` and the `Makefile` should compile it to an executable named `explorer`.

4. **The Slug**: This process randomly selects an amount of time (between 1 and 5 seconds), prints out that time length, then waits that long (it's a slug, after all). It then randomly runs either the `last -d --fulltimes` or `id -u` command (both of which exit after outputting data). This scenario's code should be in `slug.c` and the `Makefile` should compile it to an executable named `slug`.

5. **The Slug Race:** A parent process creates four children processes that will each run the `slug` executable in parallel. The Slug Race parent process will print out a note every 0.25 seconds indicating which children are still competing. When each child completes, it will report the result (with a delay of no more than just over 0.25 seconds). Once all the slug children have exited, the parent prints out a message and exits. The Slug Race parent reports on returning children as they arrive and specifically does not require them to finish in order (what kind of race would that be?). Before creating children, the Slug Race parent should get the current timestamp and, as each child completes, it should indicate how many seconds and milliseconds have elapsed since it started. This scenario's code should be in `slugrace.c` and the `Makefile` should compile it to an executable named `slugrace`.

This project will allow students to learn about process creation, termination, and resource usage in the Linux operating system. All coding is to be done in the C programming language without using any non-standard libraries. We highly recommend implementing or testing the project on a plain Ubuntu 20.04 virtual machine. Projects that do not compile or run correctly on a plain Ubuntu virtual machines without special libraries may be penalized. You **may not** use the `system()` system call in any part of Project 1 since it obscures the details involved in process creation.

These projects will be graded in bulk with some scripting support, so it is important that the associated `Makefile` produces files with the specified executable names and that the source code file names match what is specified. For the components that use randomness, students must read a number from a file, `seed.txt`, convert it from a string to an integer (using the `atoi` function from `stdlib.h`) and use it as the seed value in the `srand()` function. That will cause the seed to be used for all invocations of the `rand()` function. Examples below will be provided using the seed value `12345`, which can be used for your debugging purposes. A different seed value will be used for grading. Individual components may require additional seed files, as specified in their scenario details, and will be used in a similar fashion.

We specified the main source code file name and executable target for each of the scenarios in the earlier overview. The source code file will likely be fairly self-contained, but students may use the executable string and a hyphen as a prefix to any other source code files associated with the scenario (e.g., `slugrace-header.h`, `slugrace-helper1.c`).

We now describe each of the components in greater detail, along with their expected outputs.

## Scenario 1: The Prolific Parent

This process reads the random seed value from `seed.txt` and uses it to seed the `srand()` function as described above. It then randomly chooses the number of children using `rand()` (between 10 and 15, inclusively) and stores that value. In a loop, it then generates a random number for each child and stores it an array. Potential pitfall: It is essential that the parent construct the random numbers in advance since the `rand()` uses the seed and a global variable to track which numbers have been generated. If the random number is generated by the children after forking, they will all produce the same random number since they will have identical copies of that seed value and global variable.

Once the random number has been created, the parent then forks off the number of children determined earlier, incrementing a counter, which is initialized to 0, upon completing the loop. Each child uses the value of the counter and indexes into the array of random numbers to extract its own random value, which it uses in subsequent steps. The child obtains its own process ID and stores that value. It determines its exit code by using modulo 50 and adding one to the random number it extracted earlier (e.g., `((myRandom % 50) + 1)`) and its wait time using modulo 3 and adding one (`((myRandom % 3) + 1)`. The child prints a message revealing its own process ID, the time it will delay, and its ultimate return value. It then waits for the indicated amount of time and exits with the selected exit value.

An example transcript of Prolific Parent with seed value `12345` is below:

```
shepard@normandy:~> ./prolific
Read seed value: 12345

Read seed value (converted to integer): 12345
Random Child Count: 13
I'm feeling prolific!
[Parent]: I am waiting for PID 5924 to finish.
   [Child, PID: 5924]: I am the child and I will wait 3 seconds and exit with code 22.
   [Child, PID: 5924]: Now exiting...
[Parent]: Child 5924 finished with status code 22. Onward!
[Parent]: I am waiting for PID 5925 to finish.
   [Child, PID: 5925]: I am the child and I will wait 3 seconds and exit with code 24.
   [Child, PID: 5925]: Now exiting...
[Parent]: Child 5925 finished with status code 24. Onward!
```

```
[Parent]: I am waiting for PID 5929 to finish.
   [Child, PID: 5929]: I am the child and I will wait 2 seconds and exit with code 2.
   [Child, PID: 5929]: Now exiting...
[Parent]: Child 5929 finished with status code 2. Onward!
[Parent]: I am waiting for PID 5935 to finish.
   [Child, PID: 5935]: I am the child and I will wait 1 seconds and exit with code 22.
   [Child, PID: 5935]: Now exiting...
[Parent]: Child 5935 finished with status code 22. Onward!
[Parent]: I am waiting for PID 5936 to finish.
   [Child, PID: 5936]: I am the child and I will wait 1 seconds and exit with code 5.
   [Child, PID: 5936]: Now exiting...
[Parent]: Child 5936 finished with status code 5. Onward!
[Parent]: I am waiting for PID 5937 to finish.
   [Child, PID: 5937]: I am the child and I will wait 3 seconds and exit with code 38.
   [Child, PID: 5937]: Now exiting...
[Parent]: Child 5937 finished with status code 38. Onward!
[Parent]: I am waiting for PID 5943 to finish.
   [Child, PID: 5943]: I am the child and I will wait 2 seconds and exit with code 3.
   [Child, PID: 5943]: Now exiting...
[Parent]: Child 5943 finished with status code 3. Onward!
[Parent]: I am waiting for PID 5944 to finish.
   [Child, PID: 5944]: I am the child and I will wait 2 seconds and exit with code 29.
   [Child, PID: 5944]: Now exiting...
[Parent]: Child 5944 finished with status code 29. Onward!
[Parent]: I am waiting for PID 5945 to finish.
   [Child, PID: 5945]: I am the child and I will wait 1 seconds and exit with code 29.
   [Child, PID: 5945]: Now exiting...
[Parent]: Child 5945 finished with status code 29. Onward!
[Parent]: I am waiting for PID 5948 to finish.
   [Child, PID: 5948]: I am the child and I will wait 3 seconds and exit with code 28.
   [Child, PID: 5948]: Now exiting...
[Parent]: Child 5948 finished with status code 28. Onward!
[Parent]: I am waiting for PID 5949 to finish.
   [Child, PID: 5949]: I am the child and I will wait 1 seconds and exit with code 48.
   [Child, PID: 5949]: Now exiting...
[Parent]: Child 5949 finished with status code 48. Onward!
[Parent]: I am waiting for PID 5952 to finish.
   [Child, PID: 5952]: I am the child and I will wait 3 seconds and exit with code 22.
   [Child, PID: 5952]: Now exiting...
[Parent]: Child 5952 finished with status code 22. Onward!
[Parent]: I am waiting for PID 5953 to finish.
   [Child, PID: 5953]: I am the child and I will wait 1 seconds and exit with code 4.
   [Child, PID: 5953]: Now exiting...
[Parent]: Child 5953 finished with status code 4. Onward!
shepard@normandy:~>
```

In the above transcript, your PID values will likely differ (the OS picks those and they are dependent upon the system's history of processes). However, with the same starting seed value (`12345`), the wait times and exit codes should match and appear in the same order.

In this scenario, you must use OS system calls to `fork()` off a child process and to wait on that process. Note that the exit value of a child and wait status code are related, but a macro is needed at the parent for the values to match (consult further: `man waitpid`).

## Helpful Hints

A goal in this assignment is for students to learn how to find information in the online documentation of Unix and Linux (called `man` pages) and, from that documentation, to learn how to invoke the various system facilities from the created program. For example, to learn about the `fork()` function, type "`man fork`" in the Unix or Linux shell. Manual pages are organized into sections. Section 1 is for commands to the shell, section 2 is for system calls, and section 3 is for library routines, etc. Some entries are contained in more than one section. For example, "`man wait`" will provide the manual page for the `wait` command typed to a shell, while "`man 2 wait`" will provide the manual page for the `wait()` system call. Executing "`man man`" shows how to use the man command to view and/or print manual pages.

For this part of the assignment, the following systems calls and library functions may be needed:

- `fork()` - create a new process by cloning an existing one
- `waitpid()` - wait for a process to terminate.
- `getpid()` - get current process identifier.

## Scenario 2: Lifespan Generation

Note: This scenario is a slight modification of the Prolific Parent scenario. Students may want to start the project by copying `prolific.c` to `generation.c` and then customizing `generation.c` to implement the following.

The parent picks a random number, between 7 and 11, to represent the lifespan count. It then spawns a child. That child decrements the lifespan count, and, if the count is non-zero, creates its own child (which repeats the process) and prints out a statement indicating its process ID and that of its child. If a child decrements the count and it becomes zero (or somehow becomes negative), the child exits with an exit code of 0. All processes wait until their child returns and exits with a value that is one greater than its child's exit value. When the initial parent process confirms its child has exited, it prints out the exit value of its child and itself exits.

This scenario uses a subset of the same functions as the Prolific Parent, but rather than proceed in an iterative loop, it acts more recursively. Remember, Linux uses copy-on-write when forking, so a parent process and a child process can have different values in the same variable name if it is changed after the fork occurs.

An example transcript of Lifespan Generation with seed value 12345 is below:

```
shepard@normandy:~> ./generation
Read seed value: 12345

Read seed value (converted to integer): 12345
Random Descendant Count: 10
Time to meet the kids/grandkids/great grand kids/...
[Parent, PID: 6091]: I am waiting for PID 6092 to finish.
   [Child, PID: 6092]: I was called with descendant count=10. I'll have 9 descendant(s).
[Parent, PID: 6092]: I am waiting for PID 6093 to finish.
   [Child, PID: 6093]: I was called with descendant count=9. I'll have 8 descendant(s).
[Parent, PID: 6093]: I am waiting for PID 6094 to finish.
   [Child, PID: 6094]: I was called with descendant count=8. I'll have 7 descendant(s).
[Parent, PID: 6094]: I am waiting for PID 6095 to finish.
   [Child, PID: 6095]: I was called with descendant count=7. I'll have 6 descendant(s).
[Parent, PID: 6095]: I am waiting for PID 6096 to finish.
   [Child, PID: 6096]: I was called with descendant count=6. I'll have 5 descendant(s).
```

```
[Parent, PID: 6096]: I am waiting for PID 6097 to finish.
   [Child, PID: 6097]: I was called with descendant count=5. I'll have 4 descendant(s).
[Parent, PID: 6097]: I am waiting for PID 6098 to finish.
   [Child, PID: 6098]: I was called with descendant count=4. I'll have 3 descendant(s).
[Parent, PID: 6098]: I am waiting for PID 6099 to finish.
   [Child, PID: 6099]: I was called with descendant count=3. I'll have 2 descendant(s).
[Parent, PID: 6099]: I am waiting for PID 6100 to finish.
   [Child, PID: 6100]: I was called with descendant count=2. I'll have 1 descendant(s).
[Parent, PID: 6100]: I am waiting for PID 6101 to finish.
   [Child, PID: 6101]: I was called with descendant count=1. I'll have 0 descendant(s).
[Parent, PID: 6100]: Child 6101 finished with status code 0. It's now my turn to exit.
[Parent, PID: 6099]: Child 6100 finished with status code 1. It's now my turn to exit.
[Parent, PID: 6098]: Child 6099 finished with status code 2. It's now my turn to exit.
[Parent, PID: 6097]: Child 6098 finished with status code 3. It's now my turn to exit.
[Parent, PID: 6096]: Child 6097 finished with status code 4. It's now my turn to exit.
[Parent, PID: 6095]: Child 6096 finished with status code 5. It's now my turn to exit.
[Parent, PID: 6094]: Child 6095 finished with status code 6. It's now my turn to exit.
[Parent, PID: 6093]: Child 6094 finished with status code 7. It's now my turn to exit.
[Parent, PID: 6092]: Child 6093 finished with status code 8. It's now my turn to exit.
[Parent, PID: 6091]: Child 6092 finished with status code 9. It's now my turn to exit.
shepard@normandy:~>
```

## Scenario 3: The Explorer

"Not all those who wander are lost."
– J.R.R. Tolkien, *The Fellowship of the Ring*

In the explorer, we have a parent process that will move about the file system and will create children that execute specific commands in each location. As with the last two scenarios, the parent process will use its seed value to select a different random number for each of the five locations it will visit. The parent process will use the modulo operator with the selected random number and the number 6 (i.e., `(randomNumber % 6)`), given the 6 options, to determine which directory to change into. Below are the directories used for the six different options:

- **0:** `/home`
- **1:** `/proc`
- **2:** `/proc/sys`
- **3:** `/usr`
- **4:** `/usr/bin`
- **5:** `/bin`

For each of the five locations, it will change its current working directory (e.g., using `chdir()`), print out the option that was selected based on the random number, and output the value of the current working directory (using `getcwd()`). It then forks off a child that runs the `ls -alh` command. Done correctly, the child's listing output should match the directory the parent selected. The parent process will wait on the child to complete before moving on to the next file system location.

This scenario is most easily created using the Prolific Parent code as a starting point. Students may want to start the project by copying `prolific.c` to `explorer.c` and then customizing `explorer.c` to implement the scenario.

An example transcript of The Explorer with seed value 12345 is below:

```
shepard@normandy:~>  ./explorer
Read seed value: 12345

Read seed value (converted to integer): 12345
It’s time to see the world/file system!
Selection #1: /usr [SUCCESS]
Current reported directory: /usr
[Parent]: I am waiting for PID 11765 to finish.
   [Child, PID: 11765]: Executing ’ls -alh’ command...
total 116K
drwxr-xr-x  11 root root 4.0K Aug 26 08:08 .
drwxr-xr-x  24 root root 4.0K Jan 10 06:53 ..
[... truncated the 9 lines of output here...]
[Parent]: Child 11765 finished with status code 0. Onward!
Selection #2: /bin [SUCCESS]
Current reported directory: /bin
[Parent]: I am waiting for PID 11766 to finish.
   [Child, PID: 11766]: Executing ’ls -alh’ command...
total 17M
drwxr-xr-x  2 root root  12K Jan 14 06:25 .
drwxr-xr-x 24 root root 4.0K Jan 10 06:53 ..
-rwxr-xr-x  1 root root 1.1M Jun  6  2019 bash
[... truncated the 274 lines of output here...]
[Parent]: Child 11766 finished with status code 0. Onward!
Selection #3: /bin [SUCCESS]
Current reported directory: /bin
[Parent]: I am waiting for PID 11767 to finish.
   [Child, PID: 11767]: Executing ’ls -alh’ command...
total 17M
drwxr-xr-x  2 root root  12K Jan 14 06:25 .
drwxr-xr-x 24 root root 4.0K Jan 10 06:53 ..
-rwxr-xr-x  1 root root 1.1M Jun  6  2019 bash
[... truncated the 274 lines of output here...]
[Parent]: Child 11767 finished with status code 0. Onward!
Selection #4: /proc [SUCCESS]
Current reported directory: /proc
[Parent]: I am waiting for PID 11768 to finish.
   [Child, PID: 11768]: Executing ’ls -alh’ command...
total 4.0K
dr-xr-xr-x 313 root            root               0 Aug 26 08:14 .
drwxr-xr-x  24 root            root            4.0K Jan 10 06:53 ..
dr-xr-xr-x   9 root            root               0 Aug 26 08:14 1
[... truncated the 360 lines of output here...]
[Parent]: Child 11768 finished with status code 0. Onward!
Selection #5: /usr [SUCCESS]
Current reported directory: /usr
[Parent]: I am waiting for PID 11769 to finish.
   [Child, PID: 11769]: Executing ’ls -alh’ command...
total 116K
drwxr-xr-x  11 root root 4.0K Aug 26 08:08 .
drwxr-xr-x  24 root root 4.0K Jan 10 06:53 ..
[... truncated the 9 lines of output here...]
```

```
[Parent]: Child 11769 finished with status code 0. Onward!
shepard@normandy:~>
```

For this part of the assignment, the following new systems calls and library functions may be needed:

- `chdir()` - change the current working directory.
- `getcwd()` - get the current working directory.
- `execvp()` - execute a program (using the path environmental variable to find the executable).

## Scenario 4: The Slug

We develop The Slug scenario with Scenario 5 ("The Slug Race") in mind. The Slug will take a command line argument which can be one of four values: `1`, `2`, `3`, or `4`. Based on this value, The Slug will open files `seed_slug_1.txt`, `seed_slug_2.txt`, textttseed_slug_3.txt, or `seed_slug_4.txt`, respectively, to read in its random seed value. This allows multiple Slugs to read in different random seed values (because otherwise, all the Slugs would behave identically).

The Slug generates two random numbers. The first number represents the amount of seconds, between 1 and 5, that it will wait before proceeding. The second number represents a coin flip on whether it will run the `last -d --fulltimes` or the `id -u` command. The Slug is a single process program (i.e., it does not fork a child), so its invocation of those commands will be the end of the process.

An example transcript of The Slug with seed value `12345` stored in `seed_slug_3.txt` is below:

```
shepard@normandy:~> ./slug 3
[Slug PID: 18178] Read seed value: 12345

[Slug PID: 18178] Read seed value (converted to integer): 12345
[Slug PID: 18178] Delay time is 4 seconds. Coin flip: 1
[Slug PID: 18178] I'll get the job done. Eventually...
[ ... a delay of roughly 4 seconds occurs before the next line is output ...]
[Slug PID: 18178] Break time is over! I am running the 'id -u' command.
1001
shepard@normandy:~>
```

For this part of the assignment, the following new systems calls and library functions may be needed:

- `sleep()` - pauses execution for the specified number of seconds.

## Scenario 5: The Slug Race

This scenario is most easily created using the Prolific Parent code as a starting point. Students may want to start the project by copying `prolific.c` to `slugrace.c` and then customizing `slugrace.c` to implement the scenario.

The Slug Race parent process will spawn four children processes, each of which will run The Slug executable from Scenario 4. The parent will spawn the children using a loop with a counter, providing a different parameter to each spawned child. The parent will enter a loop, checking to see if any child has finished, and if not, waiting 0.25 seconds before printing out the current racing Slugs. The parent will continue looping until all the children have finished the race. The parent **may not** force an ordering on the children processes and it **may not** block waiting for a child to finish.

Each time a child finishes, the parent should print a message saying which child finished and how long (in seconds and milliseconds) since the start of the race that it took for the child to finish. Note that this requires the parent to have a timestamp from when the race began.

An example transcript of The Slug Race is below. Each slug has a different seed value in their respective `seed_slug_x.txt` files. These values were `55555`, `56789`, `12345`, and `456` for the `seed_slug_1.txt`, `seed_slug_2.txt`, `seed_slug_3.txt`, and `seed_slug_4.txt` files, respectively.

```
shepard@normandy:~> ./slugrace
[Parent]: I forked off child 18186.
   [Child, PID: 18186]: Executing './slug 1' command...
[Parent]: I forked off child 18187.
   [Child, PID: 18187]: Executing './slug 2' command...
[Parent]: I forked off child 18188.
   [Child, PID: 18188]: Executing './slug 3' command...
[Parent]: I forked off child 18189.
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
   [Child, PID: 18189]: Executing './slug 4' command...
[Slug PID: 18186] Read seed value: 55555
[Slug PID: 18186] Read seed value (converted to integer): 55555
[Slug PID: 18186] Delay time is 3 seconds. Coin flip: 0
[Slug PID: 18186] I'll get the job done. Eventually...
[Slug PID: 18187] Read seed value: 56789
[Slug PID: 18188] Read seed value: 12345
[Slug PID: 18187] Read seed value (converted to integer): 56789
[Slug PID: 18188] Read seed value (converted to integer): 12345
[Slug PID: 18187] Delay time is 2 seconds. Coin flip: 1
[Slug PID: 18188] Delay time is 4 seconds. Coin flip: 1
[Slug PID: 18187] I'll get the job done. Eventually...
[Slug PID: 18188] I'll get the job done. Eventually...
[Slug PID: 18189] Read seed value: 456
[Slug PID: 18189] Read seed value (converted to integer): 456
[Slug PID: 18189] Delay time is 4 seconds. Coin flip: 1
[Slug PID: 18189] I'll get the job done. Eventually...
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
The race is ongoing. The following children are still racing: 18186 18187 18188 18189
[Slug PID: 18187] Break time is over! I am running the 'id -u' command.
1001
Child 18187 has crossed the finish line! It took 2.000000 seconds
The race is ongoing. The following children are still racing: 18186 18188 18189
The race is ongoing. The following children are still racing: 18186 18188 18189
The race is ongoing. The following children are still racing: 18186 18188 18189
[Slug PID: 18186] Break time is over! I am running the 'last -d --fulltimes' command.
The race is ongoing. The following children are still racing: 18186 18188 18189
shepard  pts/1   ops.normandy.local  Thu Jan 14 14:49:07 2021   still logged in
shepard  pts/0   ops.normandy.local  Thu Jan 14 12:03:58 2021 - Thu Jan 14 14:32:45 2021  (02:28)
joker    pts/0  helm.normandy.local  Tue Jan 12 03:54:18 2021 - Tue Jan 12 03:54:24 2021  (00:00)
joker    pts/0  helm.normandy.local  Mon Jan 11 13:01:07 2021 - Mon Jan 11 15:14:19 2021  (02:13)
wtmp begins Fri Jan  1 15:42:43 2021
```

```
Child 18186 has crossed the finish line! It took 3.000000 seconds
The race is ongoing. The following children are still racing: 18188 18189
The race is ongoing. The following children are still racing: 18188 18189
The race is ongoing. The following children are still racing: 18188 18189
[Slug PID: 18188] Break time is over! I am running the 'id -u' command.
[Slug PID: 18189] Break time is over! I am running the 'id -u' command.
The race is ongoing. The following children are still racing: 18188 18189
1001
1001
Child 18188 has crossed the finish line! It took 4.000000 seconds
Child 18189 has crossed the finish line! It took 4.000000 seconds
The race is over! It took 4.000000 seconds
shepard@normandy:~>
```

For this part of the assignment, the following new systems calls and library functions may be needed:

- `usleep()` - pauses execution for the specified number of microseconds.
- `clock_gettime` - gets a high resolution clock.

## Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any assignment in which the first two scenarios are completed will be considered as making substantial progress. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

## Deliverables and Grading

When submitting your project, please include the following:

- All of the files containing the code for all parts of the assignment (e.g., `prolific.c`, `generation.c`, `explorer.c`, `slug.c`, `slugrace.c`).
- One file called `Makefile` that can be used by the `make` command for building the five executable programs. It should support the "`make clean`" command, the "`make all`" command, and be able to make each of the five programs individually.
- A document called `README.txt` explaining your project and anything that you feel the teaching staff should know when grading the project. In particular, describe the data structure and algorithm you used to keep track of background jobs. Also, explain how you tested your programs. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please **only use standard zip files** for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://ia.wpi.edu/cs3013-shue/request_teammate.php`),

2. Submit the project code and documentation via InstructAssist (URL: `https://ia.wpi.edu/cs3013-shue/files.php`),
3. Complete your Partner Evaluation (URL: `https://ia.wpi.edu/cs3013-shue/evals.php`), and

A grading rubric has been provided at the end of this specification to give you a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback. Students in teams are expected to contribute equally; unequal contributions may yield different grades for the team members.

# Project 1 – Process Party – Grading Sheet/Rubric

| | | | | |
|---|---|---|---|---|
| | | | | Evaluation? |
| Grader: | ________ | Student Name: | ________ | ________ |
| Date/Time: | ________ | Student Name: | ________ | ________ |
| Team ID: | ________ | Student Name: | ________ | ________ |
| Late?: | ________ | | | |
| Checkpoint?: | ________ | | Project Score: | / 80 |

| **Earned** | **Points** | **Task ID** | **Description** |
|---|---|---|---|
| _____ | 8 | 1 | Scenario 1 – Parent reads random seed and forks children and waits appropriately. |
| _____ | 8 | 2 | Scenario 1 – Child properly calculates delays/exit values, delays execution, and terminates. Prerequisite: Task 1. |
| _____ | 8 | 3 | Scenario 2 – Descendants forked properly with recursive waits. |
| _____ | 8 | 4 | Scenario 2 – Child processes report accurately and calculate exit codes correctly. Prerequisite: Task 3. |
| _____ | 8 | 5 | Scenario 3 – Parent process correctly changes directories, reports values using system call. |
| _____ | 8 | 6 | Scenario 3 – Child process correctly forked, executes listing in the right directory, which matches parent's output. Implements -alh parameter to ls command. Prerequisite: Task 5. |
| _____ | 8 | 7 | Scenario 4 – Command line arguments read correctly and seed file appropriately selected using argument. Delay implemented correctly. |
| _____ | 8 | 8 | Scenario 4 – Execution of commands, with all parameters, is correct. Prerequisite: Task 7. |
| _____ | 8 | 9 | Scenario 5 – Child processes execute in the background appropriately. The parent process waits on first child to return rather than imposing order. Appropriately waits on all children. |
| _____ | 8 | 10 | Scenario 5 – Timing calculation correct. Race statistics printed every 0.25 seconds while awaiting child completion. usleep correct. Prerequisite: Task 9. |

Grader Notes: