CIS 415 Operating Systems

Assignment <2> Report Collection

Submitted to:
Prof. Allen Malony

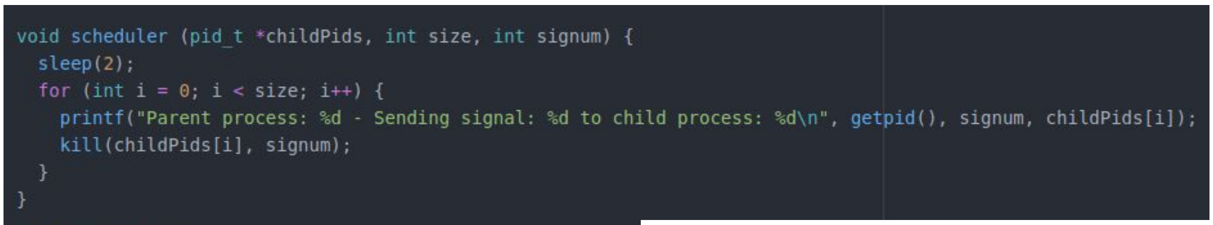
Author:
Jay (Hyo Jae) Shin

Report

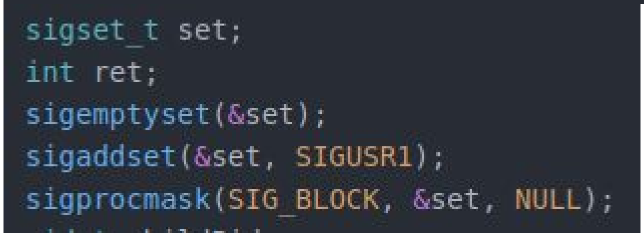
This project is working like a demo process scheduler. Providing input.txt with tasks in it, this program will check what he has to do before taking a break. First this program will tokenize and read commands distinguishing whether he can do it or not, such as invalid input in practice input. Then he will start all the tasks, yet stops all right after, because he needs SIGUSR1 to do work. Then he forks the children he can give work to. Children are getting abused by their father working valid tasks that their father handed them. While children are working, father will wait and calculate schedules to make his kids less busy.

In part1 will use read and tokenize techniques from project1 and return jumbled signals. MCP will wait until all the children are dead. Because we have exec in the child process, we don't need to worry when to end the child. using waitpid, a lazy father will have enough break time (scheduler is not implemented yet).

Then in part2, I add controlling signals, so MCP can not control children. Having sigwait function, kids will wait until father gives signal, in this case SIGCONT. I add a scheduler function, so children continue and stop their work concurrently.



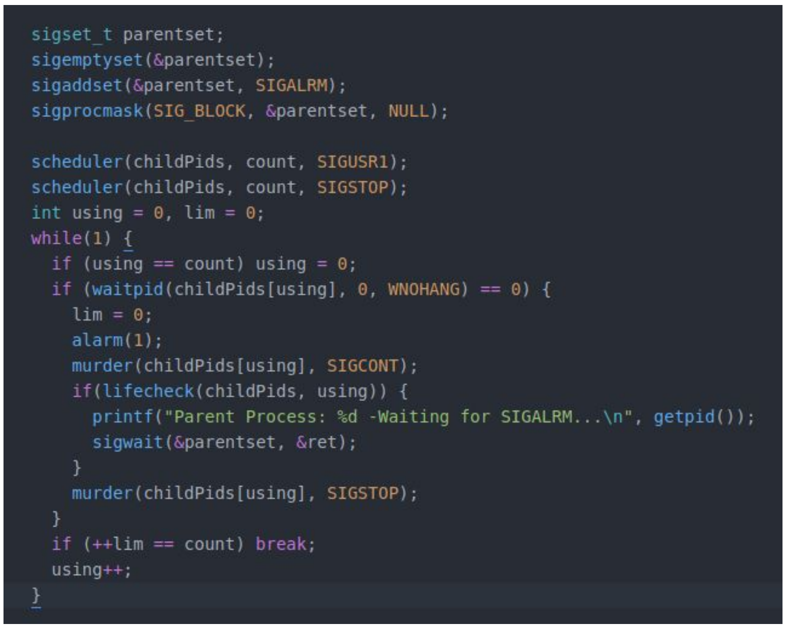
```c
void scheduler (pid_t *childPids, int size, int signum) {
  sleep(2);
  for (int i = 0; i < size; i++) {
    printf("Parent process: %d - Sending signal: %d to child process: %d\n", getpid(), signum, childPids[i]);
    kill(childPids[i], signum);
  }
}
```



```c
sigset_t set;
int ret;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL);
```

To use sigwait, we have to save signals, so I created a set of signals. After sigwait, father will just keep going through his work orders, and ready to receive updates from children.

Part3 I tried to implement the Round Robin Scheduling method. And I think I reached a good amount of work to get Part5 as well (which is to update it better). Overall structure is the same as previous steps, yet I created parent sigwait to handle SIGALRM from alarm(). I have a father waiting for a signal that the alarm set will ring in one second right before letting one of the kids to continue work. Using 2 variables to first stop the running processor, then my lifecheck function will check if in the one second nothing happen, because some very short work will end in ms. Then, it will provide all the children equally one second to finish their duties. To avoid infinite loops, I made sure that I increased both limit controls.



```c
sigset_t parentset;
sigemptyset(&parentset);
sigaddset(&parentset, SIGALRM);
sigprocmask(SIG_BLOCK, &parentset, NULL);

scheduler(childPids, count, SIGUSR1);
scheduler(childPids, count, SIGSTOP);
int using = 0, lim = 0;
while(1) {
  if (using == count) using = 0;
  if (waitpid(childPids[using], 0, WNOHANG) == 0) {
    lim = 0;
    alarm(1);
    murder(childPids[using], SIGCONT);
    if(lifecheck(childPids, using)) {
      printf("Parent Process: %d -Waiting for SIGALRM...\n", getpid());
      sigwait(&parentset, &ret);
    }
    murder(childPids[using], SIGSTOP);
  }
  if (++lim == count) break;
  using++;
}
```

Then this program results in giving all the children maximum 1 or less second every turn. Therefore this gets slightly faster than just giving all the children 1 sec no matter how they are doing (So, I assume that this fills part5 requirement). Then part4 is Part3 using /proc providing a better reading environment.

I think overall the project was better (in understanding and fun) than project1. This project was more entertaining figuring out a solution. However, I think there is too much information to look up for those who haven't had experience with low level words. Still I love this project more than the first one. Having steps to fulfill, I didn't lose interest and push myself harder to reach a solution.