Excercise Sheet 3

Johannes Koch

April 24, 2018

1 Task 1

```c
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

#define PROCESSES 9

int main(void){
  pid_t child_pid;

  // create 9 processes and store their pids
  for(int i = 0; i < PROCESSES; i++){
    // create child, exit if child or on error
    if((child_pid = fork()) == 0){
      exit(0);
    } else if (child_pid == -1){
      exit(1);
    }
  }

  // wait while the wait call returns no error, don't check exit status
    of
  // children
  while(wait(NULL) > 0);

  return EXIT_SUCCESS;
}
```

2 Task 2

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

#define PROCESSES 16

int main(void){
  pid_t pid;
  // create 16 child processes
  for(int i = 0; i < PROCESSES; i++){
    //create fork, if child print pid
    if((pid = fork()) == 0){
      // print child pid
      printf("pid: %d\n", getpid());
      exit(0);
    } else if(pid == -1){
      exit(1);
    }
  }
  /*
   * In my analysis, the order of the pid messages is generally ascending
   * altough not always perfectly ordered.
   * The parents message is normally in the middle of the other messages.
   *
   * The order cannot be predicted perfectly, other than generally the
    sooner
   * a process starts, the sooner it will finish in most cases. It
    depends on
   * many factors like the scheduling algorithm, cpu usage,...
   */
  printf("%d child processes have been created\n", PROCESSES);

  // wait for all children
  while(wait(NULL) > 0);

  return EXIT_SUCCESS;
}
```

3 Task 3

```c
#define _POSIX_SOURCE // needed for non C99 standard code like sigset_t
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/errno.h>

// use an atomic type as a flag to unblock SIGUSR2
volatile sig_atomic_t sig_block = 1;

// prints pid and signal
void confirm(pid_t pid, char* sig){
  printf("%d: received %s\n", pid, sig);
}

// signal handler for parent
void parent_handler(int sig){
  printf("%d: sending signal\n", getpid());
}

// signal handler for child
void child_handler(int sig){
  if(sig == SIGALRM){
    sig_block = 0;
    // unsafe in handlers
    printf("%d: unblocked SIGUSR2\n", getpid());
  } else if(sig == SIGUSR1){
    confirm(getpid(), "SIGUSR1");
  } else if(sig == SIGUSR2){
    confirm(getpid(), "SIGUSR2");
    // use atomic call in handler
    _exit(0);
  }
}

int main(void){
  pid_t child;
  sigset_t block_mask;

  // initialize signalset to block SIGUSR2
  sigemptyset(&block_mask);
  if(sigaddset(&block_mask, SIGUSR2) == EINVAL){
    perror("main: sigaddset");
    return EXIT_FAILURE;
  }
  // block all signals in block_mask
  if(sigprocmask(SIG_BLOCK, &block_mask, NULL) == EINVAL){
    perror("main: sigprocmask");
  }
```

```
// create signal handler
struct sigaction sa = {
  .sa_handler = parent_handler,
  .sa_flags = 0 // SA_RESTART is not supported by _POSIX_SOURCE
};
// block every signal during the handler
sigfillset(&sa.sa_mask);

// intercept SIGALRM
if(sigaction(SIGALRM, &sa, NULL) == EINVAL){
  perror("main: couldn't handle SIGALRM");
}

// create a child process
if((child = fork()) == EINVAL){
  perror("main: fork");
  return EXIT_FAILURE;
}

// child
if(child == 0){
  // overwrite the parents signalhandler
  sa.sa_handler = child_handler;
  // overwrite the signal listeners
  // intercept SIGALRM, SIGUSR1 and SIGUSR2
  if(sigaction(SIGUSR1, &sa, NULL) == EINVAL){
    perror("main: couldn't handle sigusr1");
  }
  if(sigaction(SIGUSR2, &sa, NULL) == EINVAL){
    perror("main: couldn't handle sigusr2");
  }
  if(sigaction(SIGALRM, &sa, NULL) == EINVAL){
    perror("main: couldn't handle SIGALRM");
  }

  // print PID
  printf("child PID: %d\n", getpid());

  // send SIGALRM in 15sec to unblock SIGUSR2
  alarm(15);

  // wait for signals
  for(;;){
    pause();
    // if the block_flag is unset unblock SIGUSR2
    if(!sig_block){
      // unblock the previously blocked signals
      sigprocmask(SIG_UNBLOCK, &block_mask, NULL);
    }
  }
}

// parent
```

```
  // print PID
  printf("parent PID: %d\n", getpid());

  for(int i = 0; i < 4; i++){
    // wait 5 sec (send the alarm to send a signal in 5 sec)
    alarm(5);
    pause();

    // send signal
    kill(child, i < 3 ? SIGUSR1 : SIGUSR2);
    // kill(child, SIGUSR2);
  }

  // wait for child
  waitpid(child, NULL, 0);

  return EXIT_SUCCESS;
}
```

4 Task 4

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void sigchld_handler(int signo){}

int main(void){
  // create a signal handler for SIGCHLD
  signal(SIGCHLD, sigchld_handler);
  // fork
  switch(fork()){
    case -1:
      perror("main: fork");
      exit(1);
      break;
    case 0:
      // print pid if child
      printf("%d\n", getpid());
      exit(0);
    default:
      // wait for signal if parent
      pause();
      wait(NULL);
      break;
  }
  return EXIT_SUCCESS;
}
```