COMPSCI 2SD3 Midterm Test 1
SOLUTIONS

Dr. F. Franek

DURATION : 50 minutes
McMaster University (CAS) February, 2022

Please PRINT CLEARLY your name and student number

NAME: _____

Student Number:

question	mark	out of
1-10		10
21		3
22		3
total		16

This test paper includes 10 pages, 10 multiple-choice questions, and 2 written questions. You are responsible for ensuring that your copy of the paper is complete. Bring any discrepancy to the attention of your invigilator.

Special Instructions :

1. The multiple-choice questions of this test must be answered on the form at the back of this questionnaire.
2. The written questions must be answered in the space provided in this questionnaire.
3. Documents to be returned: this questionnaire and all scrap paper if used. All of these documents must bear your name and student number. Only the face page of the questionnaire need to bear your name and student number, and all the loose pages of the questionnaire, if any.
4. No memory aids or textbooks of any kind are allowed during the test, except one letter-sized sheet of hand-written information (possibly on both sides) - this crib sheet must bear your name and student number.
5. No calculators, pocket computers, smartphones, or tablets are to be utilized.
6. No unauthorized scrap paper is allowed to be used. The invigilator(s) will supply you with needed scrap paper when you ask.
7. You are not allowed to be involved in any communication of any kind concerning the questions and answers of this test with anybody except the invigilator(s) or the instructor. Any attempt of such communication will be considered a case of academic dishonesty.

continued on next page

Questions 1 – 10 are multiple-choice questions and are to be marked on the form provided at the back of this questionnaire. For each question always select only one answer, even if you think that there are more correct answers than one. In the case that there are more correct answers, selecting one of them will earn you the full credit. Some answers may be awarded a partial credit. The negative marking is not used, i.e. incorrect or missing or multiple answers earn a 0 mark. Only the answers marked appropriately on the form at the back of this questionnaire will be considered, all other will be ignored.

Functions `sys_exit()`, `msg()`, and `msg_exit()` used in code samples below refer to `log.c` and `log.h` functions you used for Assignment 1.

Question 1 [1 mark] Consider the following C function `doit()`:

```
int doit() {
  struct sigaction oact;
  sigaction(SIGUSR1,NULL,&oact);
  return oact.sa_handler!=SIG_IGN;
}
```

Which of the following statements you think is the most appropriate?

A. This code will not compile, for there is no signal `SIG_IGN` .
B. This code will not compile, for you cannot compare signal `SIG_IGN` with a pointer to handler (`sa_handler`).
C. This code will compile, however the function `doit()` returns a random value.
⟹D. [partial credit] The function `doit()` returns 0 if the current disposition of the signal `SIGUSR1` is to be ignored, a non-zero value otherwise.
⟹E. The function `doit()` returns 0 if the current disposition of the signal `SIGUSR1` is to be ignored, 1 otherwise.

Explanation:
`doit()` returns 1 if the disposition of `SIGUSR1` is not `SIG_IGN` and 0 if it is. Thus E is correct. However D is partially correct.

Question 2 [1 mark] Consider the following program (we omitted the header files for simplification):

```
int main() {
  pid_t pid;
  int i, status, retval=1;

  for(i = 0; i < 3; i++)
   if ((pid=fork())==-1)
      sys_exit("fork failed\n");
   else if (pid==0) {
```

continued on next page

```
        msg("child %d created\n",getpid());
        break;
      }else
        msg("created child %d\n",pid);

    return 0;
  }
```

After the program was executed, we claim that we had seen the following output on the display:

```
created child 22847
child 22847 created
created child 22848
created child 22849
child 22848 created
child 22849 created
```

Is it possible?

A. No, this cannot be a real output of the program as the messages do not agree with the code.

B. No, this cannot be a real output of the program as the messages are in a wrong order.

$\Longrightarrow$C. Yes, this may be a real output.

D. Yes, this may be a part of a real output. We do not know what other messages the child processes may display, as this is not shown in the fragment of the code given to us.

E. None of the above.

Explanation:
A is wrong, since that is how the messages are coded. B is wrong, for the order of messages cannot be determined. C is right. D is wrong, for we are shown all of the code, so there are no other messages from the child process. Hence E is wrong.

Question 3 [1 mark] Choose the most appropriate answer.

A. A parent process always terminates all its child processes before its own termination.

B. A parent process must terminate all its child processes before its own termination.

$\Longrightarrow$C. A parent process may terminate all its child processes before its own termination.

D. A parent process must not terminate all its child processes before its own termination.

E. None of the above.

Explanation:
It may, but it does not have to. For instance, it is a good idea if the child processes depend on information (data) or a resource managed by the parent process. If the parent process is terminated but the child processes are left running, they would had no access to the resource or data and the system would not work properly, if at all.

continued on next page

It is not a good idea if the parent process is only used for creation of a process, but otherwise the two processes are independent. For instance, the parent process of a deamon process should terminate, the deamon process must be left running.

Question 4 [1 mark] Consider the following program (we omitted the header files for simplification) running on a single-core single CPU:

```
int volatile count=0;

void usr1_handler(int signo) {
  count++;
  msg("process %d caught SIGUSR1\n",getpid());
}
int main() {
    struct sigaction usr1_disposition;
    pid_t pid;

    sigemptyset(&usr1_disposition.sa_mask);
    sigaddset(&usr1_disposition.sa_mask,SIGUSR1);
    usr1_disposition.sa_flags=0;
    usr1_disposition.sa_handler=usr1_handler;

    sigaction(SIGUSR1,&usr1_disposition,NULL);

    if ((pid=fork())==0) {
      while(count<2) {
        kill(getppid(),SIGUSR1);
      }
    }else{
      while(count<2) {
        kill(pid,SIGUSR1);
      }
    }
   msg("%d finished\n",getpid());
   return 0;
}//end main
```

How many messages will we see on the screen when this program is executed?

A. 6 messages: two messages `...caught SIGUSR1` and one message `...finished` from both, the parent and the child
B. 5 messages, the 6 messages as in A., but the message `...finished` from the parent will be missing.
C. 5 messages, the 6 messages as in A., but the message `...finished` from the child will be missing.
D. Up to 6 messages, but how many and in what order may not be determined.
$\Longrightarrow$E. Up to any number of messages, but how many and in what order may not be determined.

Explanation:
E is correct. The first process to start after `fork()`, let us call it A will send an undetermined number of signals to the other process, let us call it B before B starts executing. But B will only get 1 signal.

continued on next page

Then B will send an undetermined number of signals to A before A starts executing. But A will only get 1. Then A again will send an undetermined number of signals to B before B starts executing. Then B starts executing and it receives the signal. Now B terminates its while loop, does not send anything to A and terminates. Now A starts to execute. It keeps sending signals to B (they will all fail), it will never receives `SIGUSR1`, so its `count` stays at 1, and A will keep continuing doing that forever.

Question 5 [1 mark] Consider the program from the previous question (question 4). Will one of the processes execute forever?

A. Yes, the parent process will never stop.

B. Yes, the child process will never stop.

$\Longrightarrow$C. Yes, one of the processes will never stop, but it cannot be determined which.

D. No, all processes will stop.

E. None of the above.

Explanation:
C is correct, see the explanation of question 4.

Question 6 [1 mark] Consider the following program (we omitted the header files for simplification):

```
pid_t myfork(pid_t pid) {
   if (pid)
     myfork(fork());
}

int main() {
   myfork(getpid());
}//end main
```

How many processes and in what "formation" will be generated by this program?

A. Just 1 process.

B. Just 2 processes, a parent and its child.

$\Longrightarrow$C. An unlimited number of process in a fan, i.e. one parent and many children.

D. An unlimited number of process in a chain, i.e. a parent, its child, and its child, etc.

E. None of the above.

Explanation:
C is correct. No child process will fork again, but the parent process will keep forking forever.

Question 7 [1 mark] Consider the following program (we omitted the header files for simplification):

continued on next page

```
int main() {
  pid_t pid;
  int i, status, retval=1;

  for(i = 0; i < 3; i++)
   if ((pid=fork())==-1)
     sys_exit("fork failed\n");
   else if (pid==0) {
     msg("child %d created\n",getpid());
     break;
   }else{
     kill(pid,SIGUSR1);
     msg("created child %d\n",pid);
   }

 return 0;
}
```

How many messages will we see on the screen when this program is executed?

$\Longrightarrow$A. At most 6 messages.
$\Longrightarrow$B. At least 3 messages.
C. Exactly 6 messages.
D. Exactly 3 messages.
E. None of the above.

Explanation:
A and B are both correct. No more than 6 messages, and no less than 3 messages. The messages of the parent will always be there (hence at least 3 messages). A message for child may or may not happen, depends on the timing when it receives the `SIGUSR1` signal.

Question 8 [**1 mark**] The UNIX system call `wait()` or `waitpid()` allow a parent process to wait for the termination of its child processes. What is the corresponding UNIX system call that allows a child process to wait for the termination of its parent process?

A. `waitcpid()`.
B. `waitchild()`.
C. `wait(pid,...)` or `waitpid(pid,...)` with a properly set parameter `pid`.
D. All of the above.
$\Longrightarrow$E. None of the above.

Question 9 [**1 mark**] The idea of switching the execution of process X to the execution of process Y is very similar to the idea of switching the execution of procedure X to procedure Y in a program. Yet totally different techniques are used and the idea of a control stack is not used for processes. Why is that so?

continued on next page

A. A control stack allows only one particular procedure to follow the execution of procedure X — the one whose activation frame is "stored" just below the one on the top. When process X "finishes" its turn on the processor, the scheduler selects the next process to follow. Which process gets selected could not be determined before the process X started and thus could not be stored on the stack.

B. A control stack allows only one particular procedure to follow the execution of procedure X — the one whose activation frame is "stored" just below the one on the top. But we usually have more than just two processes running, so we cannot use a control stack for controlling the execution of processes.

C. A control stack allows only one particular procedure to follow the execution of procedure X — the one whose activation frame is "stored" just below the one on the top. But process X can at one time execute before process Y, at some other time after, so we cannot use a control stack for for controlling the execution of processes.

$\Longrightarrow$D. Processes are scheduled according to many complex criteria, while the order of procedure calls is determined by the execution of the program itself. So when procedure Y starts executing by being called by X, it is known that the procedure X will follow when Y is finished. So a control stack can be used for procedures, while it cannot be used for processes.

E. None of the above.

Question 10 [1 mark] Consider the following program (we omitted the header files for simplification):

```
struct sigaction usr1, usr2, old;
int volatile count=0;

void usr1_handler(int signo) {
  msg("Prof. Franek is the best prof");
  count++;
  if (count < 3)
    sigaction(SIGUSR1,&usr2,NULL);
  else
    sigaction(SIGUSR1,&old,NULL);
}
void usr2_handler(int signo) {
  msg("Prof. Franek is the worst prof");
  count++;
  if (count < 3)
    sigaction(SIGUSR1,&usr1,NULL);
  else
    sigaction(SIGUSR1,&old,NULL);
}

int main() {
   usr1.sa_handler=usr1_handler;
   sigemptyset(&usr1.sa_mask);
   sigaddset(&usr1.sa_mask,SIGUSR1);
   usr1.sa_flags=0;
   usr2=usr1;
   usr2.sa_handler=usr2_handler;
```

continued on next page

```
    sigaction(SIGUSR1,&usr1,&old);

    while(count < 10)
       kill(getpid(),SIGUSR1);

  return 0;
}
```

How many and what messages will you see on the screen when this program is executed?
Despite a feeble attempt at humor, this is a real problem and a real question!

A. An infinite number of messages.
B. An infinite number of messages. First two messages will be
`Prof. Franek is the best prof`
`Prof. Franek is the worst prof`
and then it will be the message `Prof. Franek is the best prof` occurring forever
(Yes!!, choose this one).
C. An infinite number of messages. First three messages will be
`Prof. Franek is the best prof`
`Prof. Franek is the worst prof`
`Prof. Franek is the best prof`
and then it will be the message `Prof. Franek is the worst prof` occurring forever
(No way!!, don't you dare to choose this one).
$\Longrightarrow$D. Three messages only
`Prof. Franek is the best prof`
`Prof. Franek is the worst prof`
`Prof. Franek is the best prof`
E. None of the above.

Explanation:
In spite of common sense and Prof. Franek's legendary reputation dictating to choose B, D is correct. Since the program is sending the signals to itself, the counting is OK and common for both handlers. After 3 signals, the disposition of `SIGUSR1` is changed to its original (default), so the 4th signal will kill the process.

continued on next page

Questions 11 – 12 are questions that require a written answer. The answers are to be written in the space following each question.

Question 11 [3 marks] Consider the following program (listed without header files for simplicity):

```
void usr1_handler(int signo) { return; }

int main() {
  pid_t pid;
  struct sigaction usr1;
  sigset_t mask, omask;

  usr1.sa_handler=usr1_handler;
  usr1.sa_flags=0;
  sigfillset(&usr1.sa_mask);
  sigaction(SIGUSR1,&usr1,NULL);  // change SIGUSR1 disposition

  sigemptyset(&mask);
  sigaddset(&mask,SIGUSR1);
  sigprocmask(SIG_BLOCK,&mask,&omask);  // block SIGUSR1

  if ((pid = fork()) ==-1)
    sys_exit("fork failed\n");
  else if (pid==0) {
    sigprocmask(SIG_UNBLOCK,&mask,NULL);   // unblock SIGUSR1
    msg("unblocked SIGUSR1, going to pause\n");
    pause();
    msg_exit("terminating\n");
  }else{
    msg("sending signal SIGUSR1\n");
    kill(pid,SIGUSR1);
    msg_exit("terminating\n");
    pause();
  }
  return 0;
}//end main
```

This is not a safe code. In the space provided below, explain in writing why it is not safe, how the "unsafness" would demonstrate itself, and write down a corrected part of the code you think was wrong with a explanation what was wrong and how you corrected it.

Answer:

The code is not safe due to timing. If `SIGUSR1` is received by the child process before `pause()`, nothing will raise it from the `pause()`. This is remedied by an atomic action that unblocks the signal and pauses all at once: `sigsuspend()`. Only the child's code needs change:

```
  ...
  ...
else if (pid==0) {
  msg("going to unblock SIGUSR1 and pause\n");
  sigsuspend(&omask);
  msg_exit("terminating\n");
}else{
  ...
  ...
```

continued on next page

Question 12 [3 marks] Write a little C program (you do not have to put in the header files) that keeps creating a fan of child processes until the first child process terminates. Then the parent process waits for all its remaining child processes to terminate and then exits.

Each child process sleeps between 1-10 seconds in a random fashion (use `sleep((getpid()%9)+1)`) and then exits. Note that you cannot use `wait()` or `waitpid()` for waiting for the very first child process to terminate. A hint:

1. Define a global variable `quit` and set it to 0.
2. Write a handler for `SIGCHLD`. In the handler set `quit` to 1 and return `SIGCHLD` to its previous disposition.
3. In `main()`: install the new disposition for `SIGCHLD`;
4. In a loop keep forking child processes as long as `quit` is 0.
5. When the parent process leaves the loop, it waits for all its child processes and exits.
6. Each child process just goes to sleep and then exits.

A sample solution:

```
int volatile quit=0;
struct sigaction chld, old;

void chld_handler(int signo) {
  quit=1;
  sigaction(SIGCHLD,&old,NULL);
}

int main() {
  pid_t pid;
  int status, retval=1;

  chld.sa_handler=chld_handler;
  chld.sa_flags=0;
  sigaddset(&chld.sa_mask,SIGCHLD);
  sigaction(SIGCHLD,&chld,&old);

  while(!quit) {
    if ((pid=fork())==-1)
      sys_exit("fork failed\n");
    else if (pid==0) {
      sleep((getpid()%9)+1);
    }
  }

  while(retval > 0)
    while((retval=wait(&status)) == -1)
      if (errno!=EINTR) break;
}//end main
```

THE END.