Operating Systems and Computer Architecture
Final Exam — Session 1
Duration: 2 hours

Sami Taktak

January 12th, 2024

Mobile phones and other communication devices must be turned off and stored in the bags during the examination.

Make sure that you have all the pages of the subject at the beginning of the examination and report any problem of reprography if necessary.

No document allowed.

Fill your name on **every pages**.

Student's name: ______________________________

1 Numbers & Representation (5 points)

Let A, B and C 3 numbers given in binary representation.

1. compute the following $A + B = C$ operations:

①

A:		1110 0011
B:	+	0011 1001
C:	=	

②

A:		0111 0011
B:	+	0011 1001
C:	=	

③

A:		1011 1011
B:	+	1001 1011
C:	=	

④

A:		1110 0011
B:	+	0011 1101
C:	=	

First, we consider those numbers as unsigned integers:

2. What are the overflow condition on an addition between two unsigned numbers?

   ....

   ....

3. For which operation did an overflow occur?

   ....

Now, we consider those numbers as signed integers using complement's two representation:

4. What are the overflow condition on an addition between two signed numbers?

   ....

   ....

5. For which operation did an overflow occur?

   ....

2 Memory Management (5 points)

We consider a paged memory where main memory frames have a size of 1 KiB. The main memory has 16 frames numbered from 0x0 to 0xF. The memory management use an LRU algorithm.

We consider three processes A, B and C:

- The process A has an address space composed of 3 pages P0, P1 and P2. Only P0 and P2 are loaded into the main memory in frames 9 and 4 respectively;
- The process B has an address space composed of 3 pages P0, P1 and P2; Pages P0 and P1 are loaded in main memory frames 3 and 0xA respectively;
- The process C has an address space composed of 1 pages P0 loaded in main memory frame 0xF;

1. Give the format of a Page Table Entry,

2. Draw a schema representing the structure describing such a memory configuration and the main memory

3. Compute the logical and physical address of the following linear addresses and describe the process used by the processor and operating system to do the translation. In case of page miss, you will consider that the missing page is loaded in the first free page frame of the memory:
   (a) Linear Address 2448 in process A address space;

   ....
   ....
   ....
   ....
   ....

   (b) Linear Address 1116 in process B address space.

   ....
   ....
   ....
   ....
   ....

3 Write your own shell! *(10 points)*

This part is on the OSCA project *Write your own Shell!*.

1. Draw a graphical representation of the interaction of the processes as described in project architecture (see appendix A) as well as the interaction with the user. Processes and communication structures should be shown.

When the shell executes a command, it waits for the command to terminate and display its exit status. We want to extend its behaviour by:

- checking the termination status of the command and display if the command terminated normally or if it has been signaled
- if the command as exited normally display its exit value
- if the command as been signaled display signal information

`wait(2)` manual page is available in appendix B

2. Provide the code corresponding to the `wait()` call and analysis of its return value:

Now we want to add support for suspend and resume of the execution of the child command. The execution of a child process can be suspended by sending a `SIGSTOP` signal to it. Its execution could be resumed by sending a `SIGCONT` signal.

3. Provide the modification to your previous answer that are needed to provide information when these two signals are received.

A OSCA Project Architecture

The project should have the following structure:

- When started the Shell should display a prompt and wait for user input
- A user input could be a single command or a command flowed by its arguments separated by spaces
- The command should be run in a child process (`fork`)
- The child process standard input and outputs should be redirected to pipes between the child and the shell
- When the command is over the shell should display exit status of the child
- Then the shell should display a prompt again

That is, any command should run as a child process of the shell process, and the child process standards input and outputs should be redirected to the shell process using pipes. Exit status of the child process should be fetched by the shell process and displayed to the user.

B `wait(2)` manual page

Extract from Debian GNU/Linux `wait(2)` manual page.

```
wait(2)                        System Calls Manual                        wait(2)

NAME
       wait, waitpid, waitid - wait for process to change state

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <sys/wait.h>

       pid_t wait(int *_Nullable wstatus);
       pid_t waitpid(pid_t pid, int *_Nullable wstatus, int options);

DESCRIPTION
       All  of  these  system  calls  are  used  to  wait for state changes in a child of the calling
       process, and obtain information about the child whose state has changed.  A  state  change  is
       considered  to  be:  the child terminated; the child was stopped by a signal; or the child was
       resumed by a signal.  In the case of a terminated child, performing a wait allows  the  system
       to  release the resources associated with the child; if a wait is not performed, then the ter-
       minated child remains in a "zombie" state (see NOTES below).

       If a child has already changed state, then these calls return  immediately.  Otherwise,  they
       block  until  either  a  child changes state or a signal handler interrupts the call (assuming
       that system calls are not automatically restarted using the SA_RESTART flag of  sigaction(2)).
       In  the  remainder  of  this  page, a child whose state has changed and which has not yet been
       waited upon by one of these system calls is termed waitable.

   wait() and waitpid()
       The wait() system call suspends execution of the calling thread until one of its children ter-
       minates.  The call wait(&wstatus) is equivalent to:
```

```
    waitpid(-1, &wstatus, 0);

The  waitpid() system call suspends execution of the calling thread until a child specified by
pid argument has changed state.  By default, waitpid() waits only for terminated children, but
this behavior is modifiable via the options argument, as described below.

The value of pid can be:

< -1   meaning  wait  for  any  child  process whose process group ID is equal to the absolute
       value of pid.

-1     meaning wait for any child process.

0      meaning wait for any child process whose process group ID is equal to that of the call-
       ing process at the time of the call to waitpid().

> 0    meaning wait for the child whose process ID is equal to the value of pid.

The value of options is an OR of zero or more of the following constants:

WNOHANG
       return immediately if no child has exited.

WUNTRACED
       also  return  if a child has stopped (but not traced via ptrace(2)).  Status for traced
       children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
       also return if a stopped child has been resumed by delivery of SIGCONT.

(For Linux-only options, see below.)

If wstatus is not NULL, wait() and waitpid() store status information in the int to  which  it
points.   This  integer can be inspected with the following macros (which take the integer it-
self as an argument, not a pointer to it, as is done in wait() and waitpid()!):

WIFEXITED(wstatus)
       returns true if the child terminated normally, that is, by calling exit(3) or _exit(2),
       or by returning from main().

WEXITSTATUS(wstatus)
       returns the exit status of the child.  This consists of the least significant 8 bits of
       the status argument that the child specified in a call to exit(3) or _exit(2) or as the
       argument  for  a  return  statement  in  main().  This macro should be employed only if
       WIFEXITED returned true.

WIFSIGNALED(wstatus)
       returns true if the child process was terminated by a signal.

WTERMSIG(wstatus)
       returns the number of the signal that caused the  child  process  to  terminate.   This
       macro should be employed only if WIFSIGNALED returned true.

WCOREDUMP(wstatus)
       returns true if the child produced a core dump (see core(5)).  This macro should be em-
       ployed only if WIFSIGNALED returned true.

       This macro is not specified in POSIX.1-2001 and is not available on some UNIX implemen-
       tations  (e.g.,  AIX,  SunOS).   Therefore, enclose its use inside #ifdef WCOREDUMP ...
       #endif.

WIFSTOPPED(wstatus)
```

```
        returns true if the child process was stopped by delivery of a signal; this is possible
        only  if  the  call  was  done  using  WUNTRACED or when the child is being traced (see
        ptrace(2)).

    WSTOPSIG(wstatus)
        returns the number of the signal which caused the child to stop.  This macro should  be
        employed only if WIFSTOPPED returned true.

    WIFCONTINUED(wstatus)
        (since  Linux 2.6.10) returns true if the child process was resumed by delivery of SIG-
        CONT.

RETURN VALUE
    wait(): on success, returns the process ID of the terminated child;  on  failure,  -1  is  re-
    turned.

    waitpid(): on success, returns the process ID of the child whose state has changed; if WNOHANG
    was specified and one or more child(ren) specified by pid exist,  but  have  not  yet  changed
    state, then 0 is returned.  On failure, -1 is returned.

    waitid():  returns  0 on success or if WNOHANG was specified and no child(ren) specified by id
    has yet changed state; on failure, -1 is returned.

    On failure, each of these calls sets errno to indicate the error.

NOTES
    A child that terminates, but has not been waited for becomes a "zombie".  The kernel maintains
    a minimal set of information about the zombie process (PID, termination status, resource usage
    information) in order to allow the parent to later perform a wait to obtain information  about
    the  child.   As long as a zombie is not removed from the system via a wait, it will consume a
    slot in the kernel process table, and if this table fills, it will not be possible  to  create
    further  processes.   If  a parent process terminates, then its "zombie" children (if any) are
    adopted by init(1), (or by the nearest "subreaper" process as defined through the use  of  the
    prctl(2)  PR_SET_CHILD_SUBREAPER  operation);  init(1) automatically performs a wait to remove
    the zombies.

    POSIX.1-2001 specifies that if the disposition of SIGCHLD is set to SIG_IGN or  the  SA_NOCLD-
    WAIT  flag  is  set for SIGCHLD (see sigaction(2)), then children that terminate do not become
    zombies and a call to wait() or waitpid() will block until all children have  terminated,  and
    then fail with errno set to ECHILD.  (The original POSIX standard left the behavior of setting
    SIGCHLD to SIG_IGN unspecified.  Note that even though the default disposition of  SIGCHLD  is
    "ignore", explicitly setting the disposition to SIG_IGN results in different treatment of zom-
    bie process children.)

    Linux 2.6 conforms to the POSIX requirements.  However, Linux 2.4 (and earlier) does not: if a
    wait()  or  waitpid()  call  is  made while SIGCHLD is being ignored, the call behaves just as
    though SIGCHLD were not being ignored, that is, the call blocks until the  next  child  termi-
    nates and then returns the process ID and status of that child.
```