NTNU | Norwegian University of Science and Technology

Operating Systems

Discussion of PE2 – 25.02.2021

Michael Engel

2.1 Unix processes: simple alarm clock

- Write a simple alarm clock program.
- The program should ask the user to enter a number using scanf(3), which represents a delay time in seconds.
- After the number is entered, the program waits for the the given amount of time (use sleep(3)) and “rings” when the given time has passed.

- Use a loop so that after the alarm has sounded, the user is asked for a new time and a new alarm can be started.

2.1 Unix processes: simple alarm clock

```
#include <stdio.h>   // for printf/scanf
#include <stdlib.h>  // for exit(3)

int delay;

int main(void) {
  while (1) {
    printf("Enter alarm delay: ");
    scanf("%d", &delay);
    sleep(delay);
    printf("ALARM!!! \a\a\a");
  }
  exit(0);
```

Use a loop so that after the alarm has sounded, the user is asked for a new time and a new alarm can be started

The program should ask the user to enter a number using `scanf`

After the number is entered, the program waits for the the given amount of time (use `sleep(3)`)

and "rings" when the given time has passed

2.2 Multiple alarm clocks

- Support the setting of *multiple alarms* running concurrently
- After entering a delay for an alarm, create a new child process using fork(2),
  - …which is responsible for waiting the given time and then sounding the alarm.
  - When the alarm has sounded, the child should terminate using exit(3)
- While the child process is running, the parent process should already prompt the user for a new alarm delay
  - the user can set an additional alarm while a previous one is still "ticking"
- The parent should print the child process ID
- When a child process sounds an alarm, it should also print its own process ID

2.2 Multiple alarm clocks

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  // for fork()

int delay, pid;

int main(void) {
  while (1) {
    printf(“Enter alarm delay: “);
    scanf(“%d”, &delay);
    pid = fork();
    if (pid == 0) { // child process
      sleep(delay);
      printf(“ALARM from pid %d!\a\n”, getpid());
      exit(0);
    } else { // parent process
      printf(“New child: %d\n”, pid);
    }
  }
```

create a new child process using fork(2)

when child process sounds an alarm, it should also print its own process ID

the user can set an additional alarm while a previous one is still “ticking” (because it doesn’t have to wait for the child!)

parent should print the child process ID

2.2 Multiple alarm clocks

```
$ cc -o 2.2 2.2.c
$ ./2.2
Enter alarm delay: 10
New child: 95277
Enter alarm delay: 5
New child: 95280
Enter alarm delay: 2
New child: 95281
Enter alarm delay: ALARM from pid 95281!
ALARM from pid 95280!
ALARM from pid 95277!
```

Delays 10, 5 and 2 secs. were entered in fast succession.

Thus, the alarm entered last (2 sec. delay) rings first, then the middle one and finally the first one.

But – the output it garbled!

Remember: a parent process and its child processes share the same I/O channels (file descriptors).

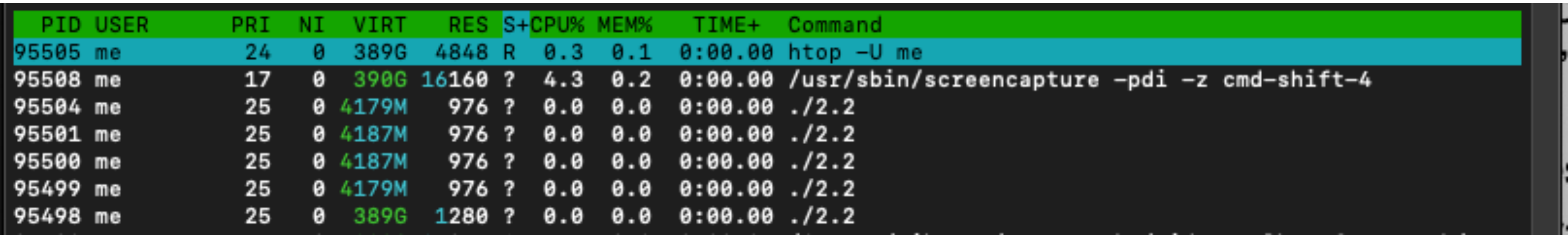
All printf() to stdout, so there can be a process switch when waiting for input at the “Enter alarm delay:” prompt.

This switches to a child process (that has finished `sleep()`ing) – this one then prints its alarm at the current cursor position on the screen.

2.3 Catch the zombies!

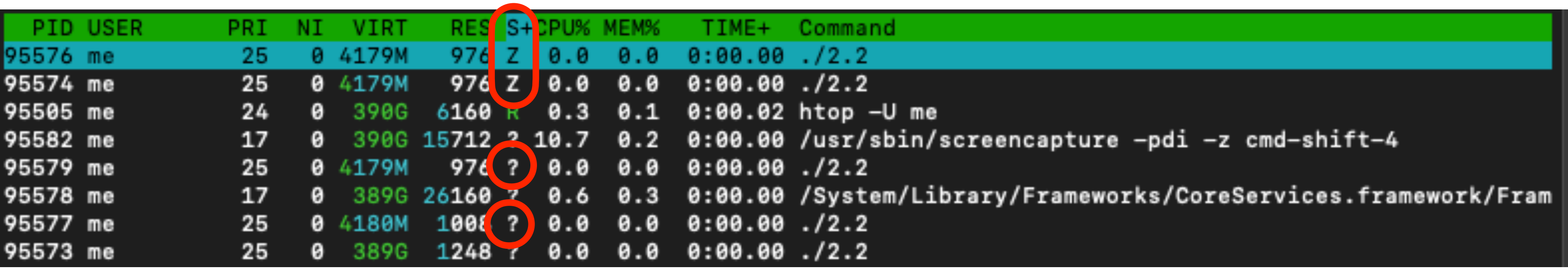
- Observe the processes started by you using the tool ps(1) or top
- You will find that the alarm clock child processes that have already rung and terminated using exit(3) are still listed as zombie processes

Better use htop – this allows to sort for process state by clicking on "S+"
All child processes (here: 4 children with 60 sec. delay) still sleeping:



```
  PID USER       PRI  NI  VIRT   RES S+CPU% MEM%   TIME+  Command
95505 me          24   0  389G  4848 R  0.3  0.1  0:00.00 htop -U me
95508 me          17   0  390G 16160 ?  4.3  0.2  0:00.00 /usr/sbin/screencapture -pdi -z cmd-shift-4
95504 me          25   0 4179M   976 ?  0.0  0.0  0:00.00 ./2.2
95501 me          25   0 4187M   976 ?  0.0  0.0  0:00.00 ./2.2
95500 me          25   0 4187M   976 ?  0.0  0.0  0:00.00 ./2.2
95499 me          25   0 4179M   976 ?  0.0  0.0  0:00.00 ./2.2
95498 me          25   0  389G  1280 ?  0.0  0.0  0:00.00 ./2.2
```

Two of four child processes have rung the alarm, the others still sleep:



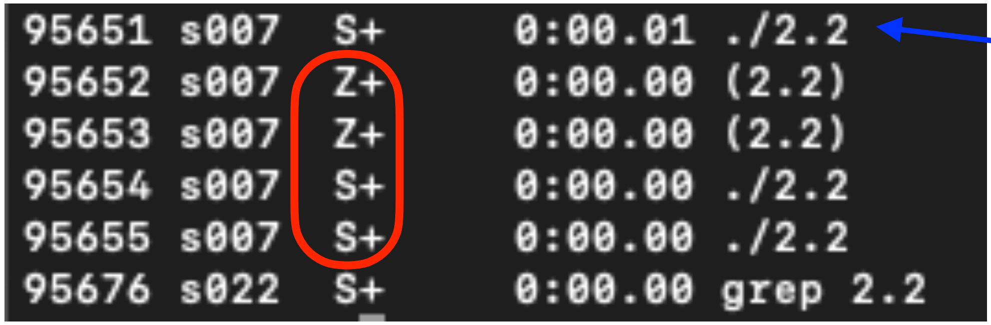
```
  PID USER       PRI  NI  VIRT   RES S+CPU% MEM%   TIME+  Command
95576 me          25   0 4179M   976 Z  0.0  0.0  0:00.00 ./2.2
95574 me          25   0 4179M   976 Z  0.0  0.0  0:00.00 ./2.2
95505 me          24   0  390G  6160 R  0.3  0.1  0:00.02 htop -U me
95582 me          17   0  390G 15712 ? 10.7  0.2  0:00.00 /usr/sbin/screencapture -pdi -z cmd-shift-4
95579 me          25   0 4179M   976 ?  0.0  0.0  0:00.00 ./2.2
95578 me          17   0  389G 26160 ?  0.6  0.3  0:00.00 /System/Library/Frameworks/CoreServices.framework/Fram
95577 me          25   0 4180M  1008 ?  0.0  0.0  0:00.00 ./2.2
95573 me          25   0  389G  1248 ?  0.0  0.0  0:00.00 ./2.2
```

2.3 Catch the zombies!

- Observe the processes started by you using the tool ps(1) or top
- You will find that the alarm clock child processes that have already rung and terminated using exit(3) are still listed as zombie processes

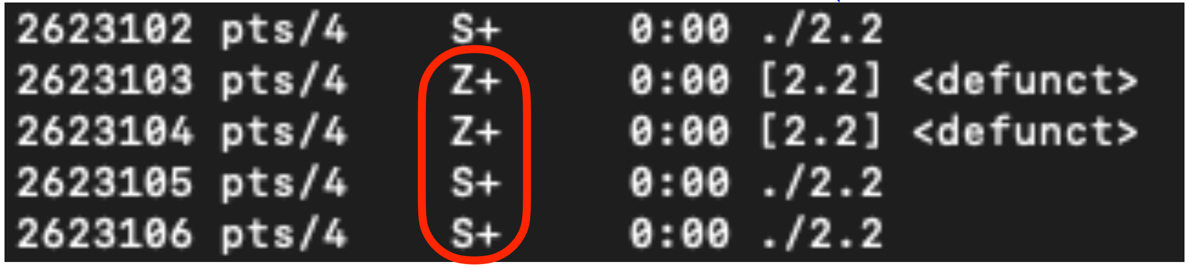
Can we get this information using ps?

```
$ ps ax | grep /2.2    # works for Linux and macOS
```



```
95651 s007  S+     0:00.01 ./2.2
95652 s007  Z+     0:00.00 (2.2)
95653 s007  Z+     0:00.00 (2.2)
95654 s007  S+     0:00.00 ./2.2
95655 s007  S+     0:00.00 ./2.2
95676 s022  S+     0:00.00 grep 2.2
```

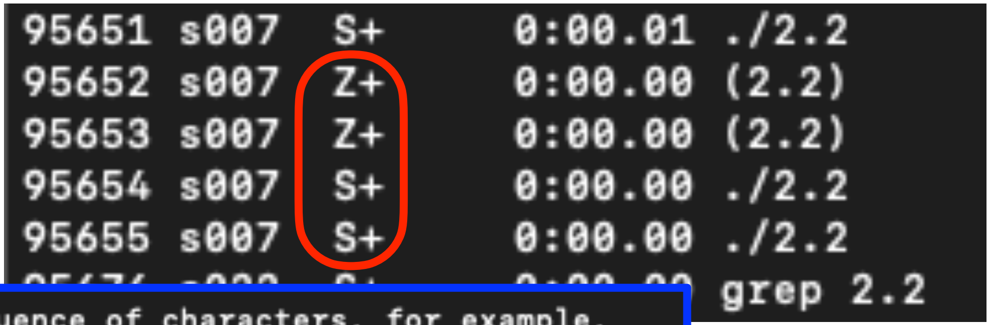
This is the parent process



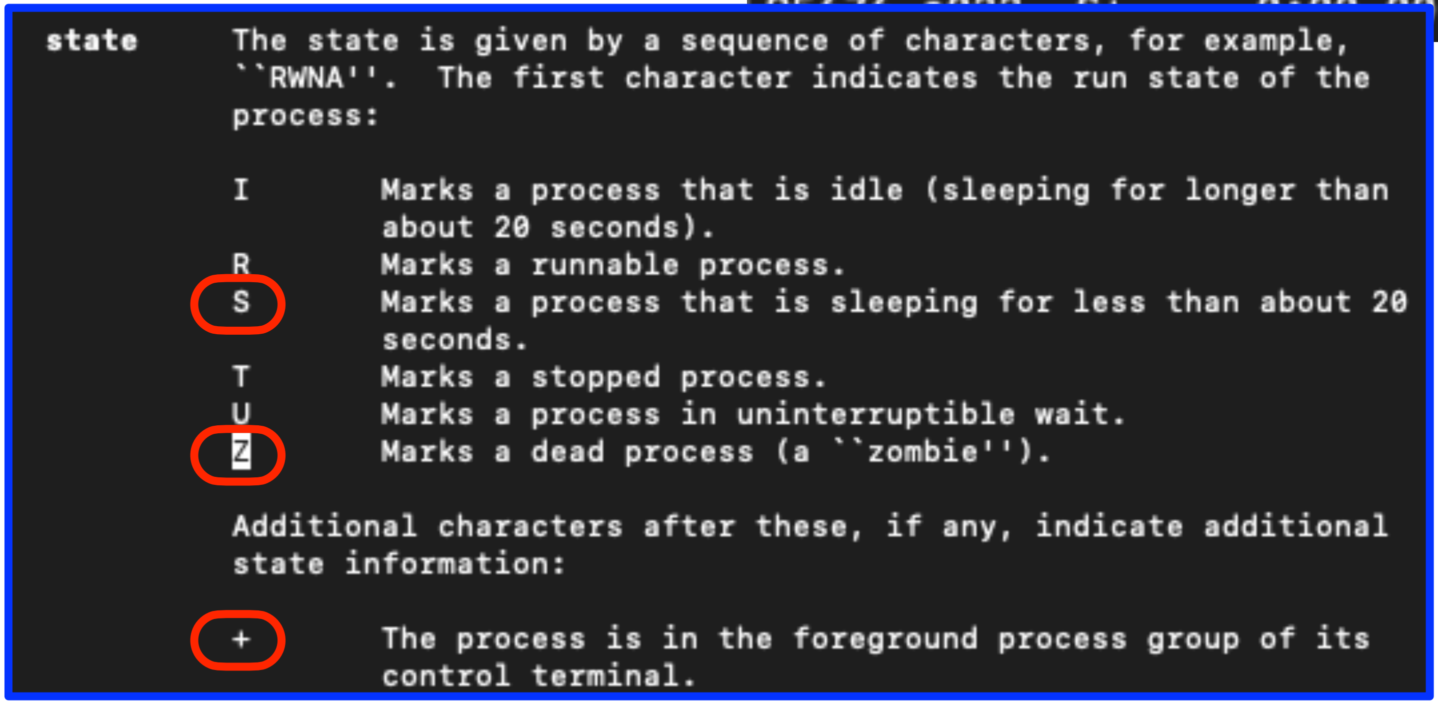
```
2623102 pts/4    S+     0:00 ./2.2
2623103 pts/4    Z+     0:00 [2.2] <defunct>
2623104 pts/4    Z+     0:00 [2.2] <defunct>
2623105 pts/4    S+     0:00 ./2.2
2623106 pts/4    S+     0:00 ./2.2
```

2.3 Catch the zombies!

- What does the output of (h)top/ps indicate? ➢ manpage…



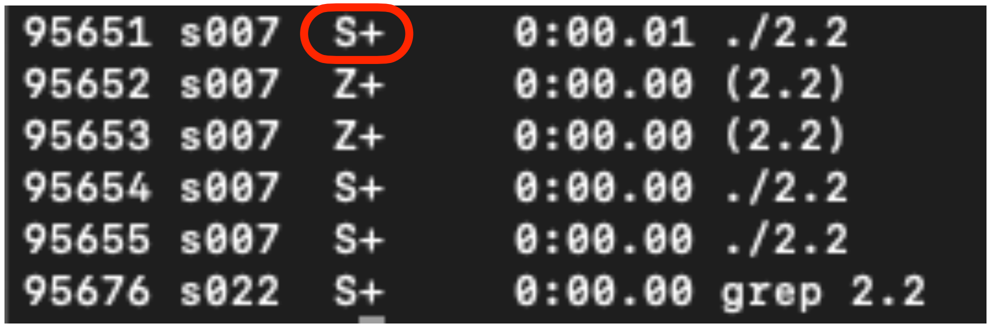
```
95651 s007  S+      0:00.01 ./2.2
95652 s007  Z+      0:00.00 (2.2)
95653 s007  Z+      0:00.00 (2.2)
95654 s007  S+      0:00.00 ./2.2
95655 s007  S+      0:00.00 ./2.2
                            grep 2.2
```



```
state     The state is given by a sequence of characters, for example,
          ``RWNA''.  The first character indicates the run state of the
          process:

          I       Marks a process that is idle (sleeping for longer than
                  about 20 seconds).
          R       Marks a runnable process.
          S       Marks a process that is sleeping for less than about 20
                  seconds.
          T       Marks a stopped process.
          U       Marks a process in uninterruptible wait.
          Z       Marks a dead process (a ``zombie'').

          Additional characters after these, if any, indicate additional
          state information:

          +       The process is in the foreground process group of its
                  control terminal.
```

2.3 Catch the zombies!

- **Why is the parent sleeping?**



```
95651 s007  S+      0:00.01 ./2.2
95652 s007  Z+      0:00.00 (2.2)
95653 s007  Z+      0:00.00 (2.2)
95654 s007  S+      0:00.00 ./2.2
95655 s007  S+      0:00.00 ./2.2
95676 s022  S+      0:00.00 grep 2.2
```

```
$ cc -o pe2.2 pe2.2.c
$ ./pe2.2
…
Enter alarm delay: ALARM from pid 95281!
ALARM from pid 95280!
ALARM from pid 95277!
```

The parent is waiting for the next input here! So it is blocked, also indicated as S

2.3 Catch the zombies!

- ***zombie processes*** remain in the system as long as the parent process does not call wait(2) or waitpid(2)
- Solve the problem of zombie processes using waitpid(2)

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf(“Enter alarm delay: “);
    scanf(“%d”, &delay);
    while ((pid = waitpid(-1, &st, WNOHANG)) > 0) {
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { // child process
      sleep(delay);
      printf(“ALARM from pid %d!\a\n”, getpid());
      exit(0);
    } else { // parent process
      printf(“New child: %d\n”, pid);
    }
  }
  exit(0);
}
```

2.3 Catch the zombies!

- ***Does this work?***

```
$ ./2.3
Enter alarm delay: 5
New child: 95978
Enter alarm delay: 5
New child: 95979
Enter alarm delay: 5
New child: 95980
Enter alarm delay: ALARM from pid 95978!
ALARM from pid 95979!
ALARM from pid 95980!
1
Child pid 95980 exited
Child pid 95979 exited
Child pid 95978 exited
```

Here, we enter a fourth alarm after the first three have already rung (and are thus *zombies* – check it!)

Note that the `waitpid()` call only catches the zombies after we have entered the next alarm delay, since `scanf()` waits for input

Some note on scanf…

- **Is the use of scanf() critical here?**

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf(“Enter alarm delay: “);
    scanf(“%d”, &delay);
    while ((pid = waitpid(-1, &st, WNOHANG))
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { // child process
      sleep(delay);
      printf(“ALARM from pid %d!\a\n”, getpid());
      exit(0);
    } else { // parent process
      printf(“New child: %d\n”, pid);
    }
  }
  exit(0);
}
```

Scanf stores the result of parsing the input in the memory address passed as parameter, so here at the memory location of the int variable `delay`.

Note that this (unless there is a bug in the kernel/libc implementation of `scanf()` is ***safe*** to do!

Some note on scanf…

- **When is scanf critical?**

```
#include <stdio.h>
int main(void) {
  int foo;
  char string[10];

  foo = 42;
  while (1) {
    printf(“String: “);
    scanf(“%s”, string);
    printf(“Entered: %s\n”, string);
  }
  exit(0);
}
```

Here, `scanf()` causes a security problem if more characters than fit in the string (-1 because terminating zero byte) are entered.

The memory locations after `string` are overwritten with the extra bytes entered!

The kernel/libc has no information about the length of the buffer for `string`, it only sees the pointer!

```
$ ./s
string at 0x0x16d5a39ae, foo at 0x0x16d5a39b8
String: Hallo
Entered: Hallo
Value of foo: 42
String: Hallo12345678901234567890
Entered: Hallo12345678901234567890
Value of foo: 959985462
```

Buffer overflow provoked!

Value of foo is changed!

2.4 Error handling

- If you read the manpages for the various system and libc calls, you will notice that there is always a section describing possible errors that are returned in case the call fails.
- Add error handling code to all system and libc calls your program makes (you can use perror(3) for this) and add code to terminate your program in case of an error.

2.4 Error handling: printf

- What can go wrong in our program? ➛ manpage…

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf(“Enter alarm delay: “);
    scanf(“%d”, &delay);
    while ((pid = waitpid(-1, &st, WNOHANG)) > 0) {
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { // child pr
      sleep(delay);
      printf(“ALARM from pid %d
      exit(0);
    } else { // parent process
      printf(“New child: %d\n”,
    }
  }
  exit(0);
}
```

```
int printf(const char * restrict format, …);

RETURN VALUES

These functions return the number of
characters printed (not including the trailing
`\0' used to end output to strings)
…
These functions return a negative value
if an error occurs.
```

2.4 Error handling: printf

- printf error handling: 2 cases
  - not all characters have been output (ret >= 0) ➛ print the rest
  - another error occured (ret < 0) ➛ complain! 😀
- We could check errors this way:

```
int len = 0;
char *s = “Delay : “;
do {
  len = printf(“%s”, s);
  if (len < 0) {
    perror(“printf”);
    exit(1);
  }
  // How many characters remain to print?
  len = strlen(s) - len;
  s += len; // increase pointer to start of string
} while (len > 0);
```

```
int printf(const char * restrict format, …);

RETURN VALUES

These functions return the number of
characters printed (not including the
trailing `\0' used to end output to strings)
…
These functions return a negative value
if an error occurs.
```

Output to the screen will usually not fail
➛ ***don’t do this*** for regular printf to screen!

For similar syscalls (e.g. write) to a file or network socket, this makes more sense…

2.4 Error handling: scanf

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf(“Enter alarm delay: “);
    scanf(“%d”, &delay);
    while ((pid = waitpid(-1, &st, WNOHANG)) > 0) {
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { /
      sleep(delay);
      printf(“ALARM !
      exit(0);
    } else { // parer
      printf(“New ch
    }
  }
  exit(0);
}
```

```
int scanf(const char * restrict format, …);

RETURN VALUES

These functions return the number of input items
assigned.  This can be fewer than provided for, or even
zero, in the event of a matching failure.

Zero indicates that, although there was input available,
no conversions were assigned; typically this is due to
an invalid input character, such as an alphabetic
character for a `%d' conversion.

The value EOF is returned if an input failure occurs
before any conversion such as an end-of-file occurs.
```

2.4 Error handling: scanf

```
int val, n;

int main() {
  do {
    printf(“Number? “);
    n = scanf(“%d”, &delay);
    if (n < 0) {
      perror(“scanf”);
      exit(1);
    }
  } while (n != 1);
  printf(“You entered: %d\n”, val);
}
```

```
$ ./val
Number? 42
Entered: 42

$ ./val
Number? qqq
Number? Number? Number? Number? Number? Number?
Number? Number? Number? Number? Number? Number?
Number? Number? Number? Number? Number? Number?
Number? Number? Number? Number? Number? Number? …
```

2.4 Error handling

```
$ ./val
Number? 42
Entered: 42

$ ./val
Number? qqq
Number? Number? Number? Number? Number?
Number? Number? Number? Number? Number?
Number? Number? Number? Number? Number?
Number? Number? Number? Number? Number?
```

- What's going on here?
  ➛ `scanf()` **could not parse the input**
  - Original incorrect input string remains in buffer
  - This string is read again in the next loop iteration!

```
int val, n;

int main() {
  do {
    printf("Number? ");
    n = scanf("%d", &delay);
    if (n < 0)  { perror("scanf"); exit(1); }
    if (n != 1) { fflush(stdin); }
  } while (n != 1);
  printf("You entered: %d\n", val);
}
```

If the number of parsed inputs is not the expected one, ***flush*** the standard input (remove all characters input so far from the input buffer)!

2.4 Error handling: waitpid

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf(“Enter alarm delay: “);
    scanf(“%d”, &delay);
    while ((pid = waitpid(-1, &st, WNOHANG)) > 0) {
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { // child process
      sl
      pr
      ex
    } el
      pr
    }
  }
  exit(0
}
```

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);

RETURN VALUES

If … waitpid() returns due to a stopped or terminated child process,
the process ID of the child is returned to the calling process.

If there are no children not previously awaited, -1 is returned with
errno set to [ECHILD].

Otherwise, if WNOHANG is specified and there are no stopped or exited
children, 0 is returned.

If an error is detected or a caught signal aborts the call, a value of
-1 is returned and errno is set to indicate the error.
```

NTN

2.4 Error handling: waitpid

```
while (1) {
  pid = waitpid(-1, &st, WNOHANG));
  if (pid < 0)  {
     if (errno == ECHILD) { printf(“ECHILD\n”); break; }
     perror(“waitpid”); exit(1);
  }
  if (pid > 0)  { printf("Child pid %d exited\n", pid); }
  if (pid == 0) { break; }  // exit the loop
}
```

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);

RETURN VALUES

If … waitpid() returns due to a stopped or terminated child process,
the process ID of the child is returned to the calling process.

If there are no children not previously awaited, -1 is returned with
errno set to [ECHILD].

Otherwise, if WNOHANG is specified and there are no stopped or exited
children, 0 is returned.

If an error is detected or a caught signal aborts the call, a value of
-1 is returned and errno is set to indicate the error.
```

2.4 Error handling: waitpid

```
while (1) {
  pid = waitpid(-1, &st, WNOHANG));
  if (pid < 0)  {
     if (errno == ECHILD) { printf(“ECHILD\n”); break; }
     perror(“waitpid”); exit(1);
  }
  if (pid > 0)  { printf("Child pid %d exited\n", pid); }
  if (pid == 0) { break; }  // exit the loop
}
```

```
$ ./2.4
Enter alarm delay: 10
ECHILD
New child: 97138
Enter alarm delay: 10
New child: 97139
Enter alarm delay: 10
New child: 97140
Enter alarm delay: 10
New child: 97141
Enter alarm delay: 10
New child: 97142
Enter alarm delay:
ALARM from pid 97138!
```

For the first iteration of the loop, no exited child is there, so we get ECHILD as return value.

This is not a ***critical error***, so we can continue running the program.

2.4 Error handling: fork

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf("Enter alarm delay: ");
    scanf("%d", &delay);
    while ((pid = waitpid(-1, &st, WNOHANG)) > 0) {
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { // child process
      sleep(delay);
      printf("ALARM fr
      exit(0);
    } else { // parent
      printf("New chil
    }
  }
  exit(0);
}
```

```
pid_t fork(void);

RETURN VALUES

Upon successful completion, fork() returns a value of 0
to the child process and returns the process ID of the
child process to the parent process.

Otherwise, a value of -1 is returned to the parent
process, no child process is created, and the global
variable errno is set to indicate the error.
```

2.4 Error handling: fork

```
pid = fork();

if (pid < 0) {
  perror(“fork”); exit(1);
}
if (pid == 0) { // child process
  sleep(delay);
  printf(“ALARM from pid %d!\a\n”, getpid());
  exit(0);
} else { // parent process
  printf(“New child: %d\n”, pid);
}
```

```
pid_t fork(void);

RETURN VALUES

Upon successful completion, fork() returns a value of 0
to the child process and returns the process ID of the
child process to the parent process.

Otherwise, a value of -1 is returned to the parent
process, no child process is created, and the global
variable errno is set to indicate the error.
```

2.4 Error handling: fork

```
pid = fork();

if (pid < 0) {
  perror("fork"); exit(1);
}
if (pid == 0) { // child process
  sleep(delay);
  printf("ALARM from pid %d!\a\n", getpid());
  exit(0);
} else { // parent process
  printf("New child: %d\n", pid);
}
```

You could also try to re-execute the `fork()` system call, since some earlier alarm processes will finish in the future.

This, however, increases the system load quite a bit…

```
$ ulimit -u 1000 # allow only 1000 user processes
$ ./2.4
ECHILD
New child: 98392
New child: 98393
… lots of lines …
New child: 99063
New child: 99064
fork: Resource temporarily unavailable
```

2.4 Error handling: sleep

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf(“Enter alarm delay: “);
    scanf(“%d”, &delay);
    while ((pid = waitpid(-1, &st, WNOHANG)) > 0) {
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { // child process
      sleep(delay);
      printf(“ALARM from pid %d!\a\n”, getpid());
      exit(0);
    } else { // parent process
      pri
    }
  }
  exit(0)
}
```

```
unsigned int sleep(unsigned int seconds);

RETURN VALUES

If the sleep() function returns because the requested time has
elapsed, the value returned will be zero.

If the sleep() function returns due to the delivery of a signal, the
value returned will be the unslept amount (the requested time minus
the time actually slept) in seconds.
```

2.4 Error handling: sleep

```
if (pid == 0) { // child process
  int unslept = delay;
  do {
     unslept = sleep(unslept);
  } until (unslept > 0);

  printf(“ALARM from pid %d!\a\n”, getpid());
  exit(0);
} else { …
```

```
unsigned int sleep(unsigned int seconds);

RETURN VALUES

If the sleep() function returns because the requested time has
elapsed, the value returned will be zero.

If the sleep() function returns due to the delivery of a signal, the
value returned will be the unslept amount (the requested time minus
the time actually slept) in seconds.
```

2.4 Error handling: exit

```
// …includes omitted…
int delay, pid, st;

int main(void) {
  while (1) {
    printf("Enter alarm delay: ");
    scanf("%d", &delay);
    while ((pid = waitpid(-1, &st, WNOHANG)) > 0) {
      printf("Child pid %d exited\n", pid);
    }
    pid = fork();
    if (pid == 0) { // child process
      sleep(delay);
      printf("ALARM from pid %d!\a\n", getpid());
      exit(0);
    } else { // parent process
      printf("New child: %d\n", pid);
    }
  }
  exit(0)
}
```

```
void exit(int status);

RETURN VALUES

The exit() and _Exit() functions never return.
```

Accordingly, we cannot handle any errors. So a program will always terminate successfully on Unix!

2.4 Discussion: error handling

- This task only gave ***a single puny point***… why?
  - I wanted you to figure out the multitude of different errors that can actually occur in a Unix process
  - …and there are a lot!

- Correct and complete error handling causes significant overhead in code
  - Unfortunately, almost nobody does this!
  - C does not have a facility for ***exception handling***
    - so error handling is a horrible kludge
  - C does not support ***multiple return values*** for functions
    - e.g. Go allows `f,err := os.Open("filename.ext")`