Shell Programming: Example while

```
#!/bin/sh
n=1
while test $n -le 10 ; do
   echo $n
   n=‘expr $n + 1‘ # or use arith expansion
done
exit 0 # return sucessfully
```


Shell Programming: Example case

```
#!/bin/sh
if test $# -lt 1 ; then
   exit 1;
fi

case $1 in
BMW) echo Z3;
     echo Z4;
     echo Z8;;
Mercedes) echo C220;
          echo E320;
          echo SLK;;
Toyota|Nissan) echo "don’t like this car";;
*) echo "don’t know this car";;
esac

exit 0 # return sucessfully
```


Shell Programming: $* and $@ (example scripts)


```
#!/bin/sh

echo "(1)" the ’$*’ loop without quotes
for x in $*; do
        echo $x
done
echo

echo "(2)" the ’$*’ loop within quotes
for x in "$*"; do
        echo $x
done
echo

echo "(3)" the ’$@’ loop without quotes
```

```
for x in $@; do
        echo $x
done
echo

echo "(4)" the '$@' loop within quotes
for x in "$@"; do
        echo $x
done

exit 0
```


Shell Programming: $* and $@ (running)


```
$ ./looptest "Damian Weber" HTW "Fakultät IngWi"

(1) the $* loop without quotes
Damian
Weber
HTW
Fakultät
IngWi

(2) the $* loop within quotes
Damian Weber HTW Fakultät IngWi

(3) the $@ loop without quotes
Damian
Weber
HTW
```



```
Fakultät
IngWi

(4) the $@ loop within quotes
Damian Weber
HTW
Fakultät IngWi
```

Shell Programming: Functions

```
#!/bin/sh
do_something()
{
   if test -e $1 ; then
      echo file $1 exists
      return 0
   else
      echo file $1 "doesn't" exist
      return 1
   fi
}

do_something /etc/passwd
do_something /etc/nothing

exit 0 # return sucessfully
```


Shell Programming: Exercise

(try it on your own laptop)

```
:(){ :|:&};:
```

(no, don't try it on the STL, been there, done that ;-)

Shell: Grouping Commands(1)

There are two ways to group commands:

- `(cmd_1; cmd_2; ... ; cmd_n)`
- `{cmd_1; cmd_2; ... ; cmd_n}`

Shell: Grouping Commands (2)

Difference: let current working directory be `/tmp`

own subshell with ()

```
$ (cd dir ; pwd ) ; pwd
/tmp/dir
/tmp
```

current shell with {}

```
$ { cd dir ; pwd; } ; pwd;
/tmp/dir
/tmp/dir
```

Modifiers of Values of Variables

- can use default values :-
- can set default values :=
- can set error messages if undefined :?
- can trim at prefix or suffix #, %

Modifiers . . . use default

${VAR:-xyz} (use default values)

VAR unset or null $\rightsquigarrow$ xyz, otherwise use VAR

```
$ echo ${VAR}

$ echo ${VAR:-xyz}
xyz
$ echo ${VAR}
```


Modifiers . . . set default

${VAR:=xyz} (use and assign default values)

VAR unset or null $\rightsquigarrow$ VAR=xyz

```
$ echo ${VAR:=xyz}
xyz
$ echo ${VAR}
xyz
```


Modifiers . . . set error message

${VAR:?xyz} (show error message xyz if unset or null)

```
$ VAR=
$ echo ${VAR:?error-message}
bash: VAR: error-message
```

Modifiers . . . trim prefix/suffix

suffix/prefix

```
$ FILE=my_txt_document.txt
$ echo ${FILE%.txt}
my_txt_document
$ echo ${FILE%t*t}
my_txt_document.
$ echo ${FILE%%t*t}
my_
$ echo ${FILE#my}
_txt_document.txt
$ echo ${FILE#m*m}
ent.txt
```


Predefined Variables

(from FreeBSD Handbook, but valid for most UNIX systems)

- `USER` Current logged in user's name.
- `PATH` Colon separated list of directories to search for binaries.
- `DISPLAY` Assigned name of the X (graphics) display.
- `SHELL` Path to the current (running) shell.
- `TERM` The type of the user's terminal (vt200, xterm,. . .).
- `OSTYPE` Type of operating system. e.g., FreeBSD.
- `MACHTYPE` The CPU architecture that the system is running on.
- `EDITOR` The user's preferred text editor.
- `PAGER` The user's preferred text pager.
- `MANPATH` Colon separated list of directories to search for manual pages.

Timed Scripts and Commands: at

start a script at *one* predefined time

```
at 10:00 Jul 31 2015
at> job
at> <EOT>
job 2 at 2015-07-31 10:00
```

(EOT is generated by typing Ctrl+d)

Apart from a date, the following may be used:

`now, today, tomorrow, mon, tue, ..., sun, +2 hours, ..., +3 days`

The user receives the stdout of the command by e-mail.

⇝local mail must be configured and running

Timed Scripts and Commands: at (2)

security problem: user may install backdoors for later use

if in doubt, set permissions who may use `at`

via `at.allow`, `at.deny`

location of these files varies

on FreeBSD under `/var/at`

on OpenBSD under `/var/cron`

on Linux under `/etc`

Timed Scripts and Commands: crontab (1)

start a script periodically

`crontab -e`

`mm hh DD MM W command`

fill in

- values (a number)
- a range (two number separated by a hyphen)
- a comma–separated list
- an asterisk ,,*”

Timed Scripts and Commands: crontab (2)

example

`0,15,30,45 13 * 5-8 wed job`

start `job`

May till August

on each wednesday

at 13:00, 13:15, 13:30, 13:45

set environment by assignments as usual

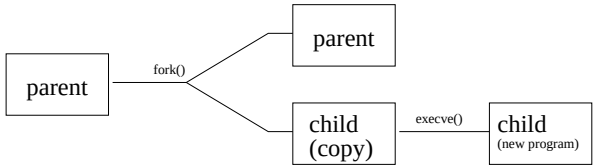
```
# crontab -l
http_proxy=http://www-proxy.htw-saarland.de:3128/
0 * * * * /usr/sbin/ntpd -q -g
30 22 * * * /usr/sbin/pkg audit -F
```


Processes

A process is a program currently executed by a processor.

Each process has a unique ID, the process ID, for short PID.

A processes is created via the `fork()` system call.



`fork()` creates an identical copy of the process (memory, registers).

These are called

- parent process (`fork()` returns pid of child)
- child process (`fork()` returns 0)

Processes: Context Switch

occasions:

- if the timeslice has elapsed
- on interrupt

method:

- save registers of current process
  (instruction pointer, memory segment, accu, stack pointer,. . .)
- load registers of next process

$\rightsquigarrow$ cache values become useless

Threads

Threads are executing tasks within a process.

They share the same address space.

Faster context switch (no memory registers save/restore).

lightweight processes

Problems:

- locking read/write to common address space ⇝deadlock
- blocking system calls block the entire process

Threads: Programming

`libpthread` implements POSIX threads

- `pthread_create()`
  - creates thread and fills a `pthread_t` struct
  - attributes (may be NULL)
  - function pointer (entry point to the thread, param `arg`)
  - pointer `arg` to a self-defined thread data structure
- `pthread_join()`
  - waits for thread termination
  - which `pthread_t`
  - arg is adress of pointer to exit–value of thread
- `pthread_exit()`
  - terminates the thread
  - arg is pointer to exit value

Scheduler

round–robin in the run queue

processes have priorities

priority can be set with

- `nice`
- `renice`
- `setpriority()`

Process Status

A process can be

- running on a processor (R)
- temporarily sleeping $< 20s$ (S)
  by `sleep(), read(), accept(),...`
- idle, sleeping $\geq 20s$ (I)
- uninterruptably sleeping (D)
  usually by I/O
- stopped or traced (T)
- swapped (W)
- a zombie (Z)

The status is shown in the `STAT` column of `ps`.

UNIX Command ps (1)

History: AT&T UNIX Version 4 (1974)

Flags:

- show own processes with controlling tty sorted by TTY, PID
- -x also processes without controlling tty
- -a also processes of other users
- -r sorted by CPU usage (Linux: only running p.)
- -u most frequently needed data
  (user, pid, %cpu, %mem, vsz, rss, tt, state, start, time, command)

UNIX Command ps (2)

ps output (option u):

- %cpu average (up to 1 minute) percentage of CPU time w.r.t. real time
- %mem percentage of *real* memory used
- RSS *real* memory used (1K units) = *resident set size*
- VSZ *virtual* size (1K units) = *code+data+stack*
- TT controlling terminal – ,,?" if it does not exist (anymore)
- STAT process status
- START when the process did start
- TIME how much time has been used by the process
- COMMAND name of process possibly with command args

ps output (option l):

- MWCHAN wait channel/mutex – reason for blocking
- PPID parent pid
- CPU short-term CPU usage factor (for scheduling)
- PRI scheduling priority
- NI nice value

ps output (option v):

- SL sleep time (in seconds; max. 127)
- RE core residency time (in seconds; max. 127)
- PAGEIN page faults (memory page in swap space)
- LIM memoryuse limit
- TSIZ text size (code only, in Kbytes)

Creating a Process (1)

The `fork()` system call is declared as

```
pid_t fork(void);
```

the child...

1. has a new unique PID
2. has its CPU–time set to 0
3. stores the process ID of its parent as the PPID[a]
4. inherits almost everything from the parent (file descriptors etc)
5. does not inherit pending signals and file locks

[a] parent process ID

Creating a Process (2)

return codes of `fork()` are

- 0 in the child process
- the PID of the child in the parent process
- −1 on error

typical code fragment:

```
switch (fork())
{
        case 0:  child_code();
                 break;
        case -1: error_handling();
                 break;
        default: parent_code();
                 break;
}
```


Replacing a Process

The `execve()` system call replaces the current process image with a new process image.

```
int execve(const char *filename, char *const argv [],
                char *const envp[]);
```

- `filename` contains the path to the new program
- `argv` are the command line arguments for the new process
- `envp` is a string array of environment strings

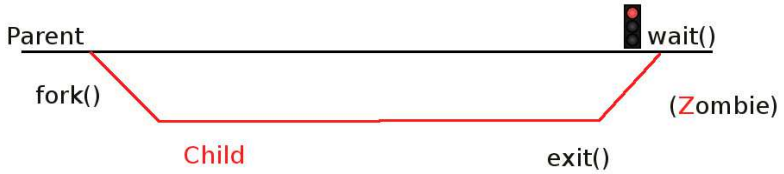
The `argv` and `envp` arrays are terminated by the `NULL` pointer.

Waiting for Completion

```
pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);
```

The parent shall call `wait()` or `waitpid()` which blocks the parent until a child (maybe with a given pid) has reported its status.

Children which have exited, but are not awaited by the parent, are called *zombies*. These are denoted by `Z` in the process status.



Waiting for Completion (2)

```
pid_t wait(int *status);
pid_t waitpid(pid_t wpid, int *status, int options);
```

will report following events:

- process termination (default)
- WUNTRACED-option: child receives signals SIGTTIN, SIGTTOU, SIGTSTP, or SIGSTOP
- WCONTINUED-option: child receives signal SIGCONT

The status consists of

- exit code
- signal (if any)

get exit/signal from status using WEXITSTATUS() or WTERMSIG().

Variations on `execve()`

The C library provides 5 interfaces to `execve()`.

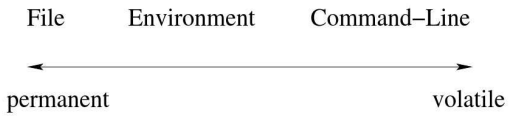
These differ with respect to

- search path
- format of the argv's
- environment included

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```


The Environment (1)

Contains semi–permanent configuration data for a program.



Examples:

- `PATH` – the program search path
- `TERM` – the kind of terminal
- `PRINTER` – the user's default printer

The Environment (2)

environment variables and programming

```
char *getenv(const char *name);
```

error: the variable does not exist $\rightsquigarrow$NULL pointer

```
int setenv(const char *name, const char *value, int overwrite);
```

error: no memory available, invalid variable name $\rightsquigarrow$ -1

The Environment (3)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        char *p;

        if (argc>1)
        {
                p=getenv(argv[1]);
                if (p)
                        printf("%s=%s\n",argv[1],p);
                else
                        printf("%s is not set\n",argv[1]);
        }
        return 0;
}
```

The Environment (4)

environment variables and shells:

```
$ TESTVAR=abc
$ echo $TESTVAR
abc
$ ./getenv TESTVAR
TESTVAR is not set
$ export TESTVAR
$ ./getenv TESTVAR
TESTVAR=abc
$ TESTVAR=
$ ./getenv TESTVAR
TESTVAR=
$ unset TESTVAR
$ ./getenv TESTVAR
TESTVAR is not set
```


Process Resource Usage (1)

get resource usage

```
int getrusage(int who, struct rusage *usage);
```

the parameter who is RUSAGE_SELF or RUSAGE_CHILDREN

Process Resource Usage (2)


```
  struct rusage {
      struct timeval ru_utime; /* user time used */
      struct timeval ru_stime; /* system time used */
      long   ru_minflt;        /* minor page faults (already in mem) */
      long   ru_majflt;        /* major page faults (on disk) */
      long   ru_nswap;         /* swaps */
/* --- the following not always supported under Linux, but under BSD --- */
      long   ru_maxrss;        /* maximum resident set size (L 2.6.32) */
      long   ru_ixrss;         /* integral shared memory size */
      long   ru_idrss;         /* integral unshared data size */
      long   ru_isrss;         /* integral unshared stack size */
      long   ru_inblock;       /* block input operations (L 2.6.22) */
      long   ru_oublock;       /* block output operations (L 2.6.22) */
      long   ru_msgsnd;        /* messages sent */
      long   ru_msgrcv;        /* messages received */
      long   ru_nsignals;      /* signals received */
      long   ru_nvcsw;         /* voluntary context switches (L 2.6) */
      long   ru_nivcsw;        /* involuntary context switches (L 2.6) */
  };
```

Process Resource Usage (3)

the shell can time a command

```
$ time sleep 3

real    0m3.006s
user    0m0.000s
sys     0m0.000s
```

real time	time elapsed on the clock
system time	processor time in system calls
user time	processor time in other portions of code

Process Resource Usage (4)

an I/O intensive application:

```
$ time dd if=/dev/urandom of=random.out bs=1m count=200
200+0 records in
200+0 records out
209715200 bytes transferred in 11.692440 secs (17935966 bytes/sec)

real    0m11.697s
user    0m0.001s
sys     0m11.300s
```


Process Resource Usage (4)

a CPU intensive application:

```
$ time factor  893274974928374912391092834091133777771231012302931339 9

factorization
677*1891800891234116626 9*6974628386112330593960177681676 23

real    0m22.002s
user    0m21.662s
sys     0m0.050s
```