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

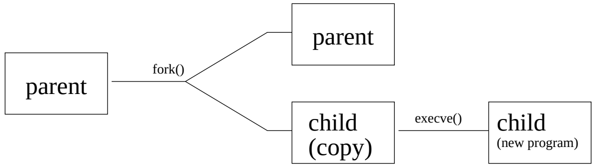
```
# crontab -l
http_proxy=http://www-proxy.htw-saarland.de:3128/
0 * * * * /usr/sbin/ntpd -q -g
30 22 * * * /usr/sbin/pkg audit -F
```

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

⇝ 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()`.

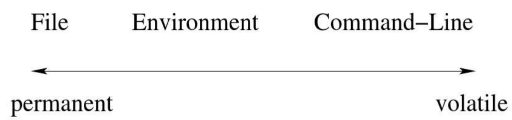
These differ with respect to

- 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 ⇝NULL pointer

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

error: no memory available, invalid variable name ⇝–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 are not supported under Linux  but under BSD --- */
    long   ru_maxrss;        /* maximum resident set size */
    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 */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* messages sent */
    long   ru_msgrcv;        /* messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```


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  8932749749283749123910928340911337777712310123029313399

factorization
677*18918008912341166269*697462838611233059396017768167623

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


Semaphores

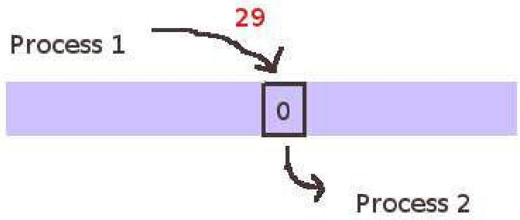
an IPC mechanism

inter-process communication

needed if two processes share a common resource, primarily memory

shared memory

Shared Memory Problem



assume value 0 in adress 0x10000000

Process 1 writes value 29 to address 0x10000000

Process 2 reads from address 0x10000000

when process 2 reads from 0x10000000, does it read a 0 or a 29 ?

Problem

- perhaps process 1 was stopped
- perhaps process 2 was stopped
- perhaps one of them runs at lowest priority
- perhaps one of them delayed because of a I/O problem
- ...

process 2 must be stopped before reading until process 1 has written

Visualization of Semaphores

wait until memory is updated...

I want to read...
(P–Operation)

I am allowed to read...
(someone did V–Operation)

Theory of Semaphores

invented by Dijkstra 1968

http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html

critical section: only one process is allowed to enter CS

P–Operation: (dutch ,,passeren")

- process wants to enter CS,
- but is blocked if some other process in CS
- in CS, process allocated the resource

V–Operation: (dutch ,,vrijgeven")

- process leaves CS,
- releases resource

UNIX: Semaphore Set

a vector of n semaphores comprise a semaphore set

semaphore: (semaphore ID, semaphore number)

obtain a semaphore set by `semget()`

operations on semaphore set by `semop()` : **P**, **V**

remove semaphore set by `semctl()`