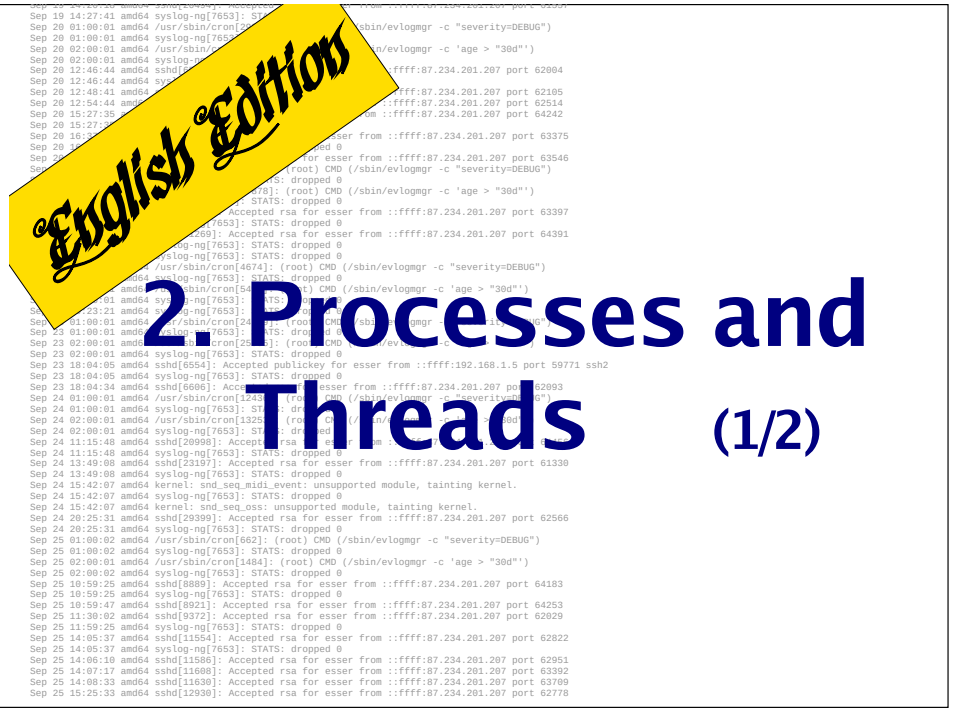



# 2. Processes and Threads (1/2)

## Processes & Threads: Outline

**Introduction**

**Practice I:** Processes in the shell
Processes in C programs

**Theory:** Processes and threads

**Practice II:** Threads in C programs

**Practice III:** Processes and threads in the Linux kernel



## Introduction (1)

**Single-Tasking / Multitasking:**

How many programs run simultaneously?

- MS-DOS, CP/M: 1 program
- Windows, Linux, ...: many programs

**Single-Processing / Multi-Processing:**

Does it help to use several CPUs?

- Windows 95/98/Me: 1 CPU
- Windows 2000, XP, Linux, Mac OS X, ...: Several CPUs



## Introduction (2)

**MS-DOS:**

- Operating system starts, activates shell COMMAND.COM
- User enters command
- If it is no internal command: load and activate program
- After end of program execution: Jump back to COMMAND.COM

So switching between several programs

# Introduction (3)

**Process:**

- Need a concept, once we have >1 program running
- program that is to be executed by the machine
- Its own data
- sealed off from other processes
- additional administrative data




# Introduction (4)

**Process list:**

- Information about all processes and their states
- Each process has a **Process Control Block** (**PCB**):
  - Identifier (PID)
  - Register values incl. program counter (instruction counter)
  - memory area of the process
  - Lists of open files and sockets
  - Information such as parent PID, last activity, total running time, priority, ...



# Practice: User (1)

```
esser@sony:Skript> emacs test.txt &
[3] 24469
esser@sony:Skript> _

[...]

[3]+ Done                    emacs test.txt
```

# Practice: User (2)

```
esser@sony:Skript> jobs
[1]- Running                 xpdf -remote sk skript-bs.pdf &
[2]+ Running                 nedit kap02/index.tex &

esser@sony:Skript> jobs -l
[1]- 8103 Running            xpdf -remote sk skript-bs.pdf &
[2]+ 20568 Running           nedit kap02/index.tex &

esser@sony:Skript> ps w|grep 8103|grep -v grep
 8103 pts/15 S       5:27 xpdf -remote sk skript-bs.pdf
```



# Practice: User (3)

```
> ps auxw
USER    PID %CPU %MEM    VSZ   RSS TTY    STAT START   TIME COMMAND
root      1  0.0  0.0    720    92 ?      S    Jun24   0:01 init [5]
root      2  0.0  0.0      0     0 ?      SN   Jun24   1:09 [ksoftirqd/0]
root      3  0.0  0.0      0     0 ?      S<   Jun24   0:11 [events/0]
root      4  0.0  0.0      0     0 ?      S<   Jun24   0:00 [khelper]
root      5  0.0  0.0      0     0 ?      S<   Jun24   0:00 [kthread]
root      7  0.0  0.0      0     0 ?      S<   Jun24   0:02 [kblockd/0]
root      8  0.0  0.0      0     0 ?      S<   Jun24   0:00 [kacpid]
root    128  0.0  0.0      0     0 ?      S<   Jun24   0:00 [aio/0]
[....]
esser  5733  0.2 12.2  82420 63428 ?      S    Jul24   4:05 /usr/lib/opera/opera
root   2670  0.3  0.0   1368   300 ?      Ss   08:24   2:39 zmd /usr/lib/zmd/zmd.exe
esser  8037  0.0  0.6   6452  3384 pts/13 S+   11:23   0:05 ssh -X amd64
```



# Practice: User (4)

```
> pstree -p
init(1)-+-acpid(2266)
        |-auditd(2727)---{auditd}(2728)
        |-cron(3234)
        |-cupsd(2706)
        |-gpg-agent(4031)
        |-hald(2309)-+-hald-addon-acpi(2616)
        |            |-hald-addon-stor(2911)
        |            `-hald-addon-stor(2914)
        |-kded(4079)
        |-kdeinit(4072)-+-artsd(7184)
        |               |-kio_file(4402)
        |               |-klauncher(4077)
        |               |-konqueror(22430)
        |               |-konsole(11064)-+-bash(11065)---ssh(31205)
        |               |                |-bash(11119)---sux(11444)---bash(11447)
        |               |                |-bash(11137)
        |               |                |-bash(25637)-+-ssh(4522)
        |               |                |             `-xmms(7169)-+-{xmms}(7170)
        |               |                |                          `-{xmms}(7171)
        |               |                `-bash(15608)
        |               |-konsole(4773)-+-bash(4774)---ssh(8037)
        |               |               |-bash(8040)---ssh(8058)
        |               |               `-bash(8061)-+-less(15188)
        |               |                             |-nedit(9628)
        |               |                             `-xpdf(8103)
```



# Practice: User (5)

- suspend program: Strg-Z
- continue in foreground: **`fg`**
- continue in background: **`bg`**
- send signal to process: **`kill`**
  - suspend (STOP), continue (CONT)
  - terminate (TERM), destroy (KILL)
- detach child from parent: **`disown`**

# Practice: User (6)

```
> kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```



# Practise: Programmer (1)

Create new process: **`fork ()`**

```
main()
{
  int pid=fork();   /* Create child process */
  if (pid == 0)
     {
       printf("I'm the child, my PID is %d.\n", getpid() );
     }
  else
     {
       printf("I'm the parent, my child has PID %d.\n", pid);
     }
}
```



# Practise: Programmer (2)

Execute a different program in the new process: **`exec ()`**

```
main()
{
  int pid=fork();   /* Create child process */
  if (pid == 0)
     {
       /* child launches external program */
       execl( "/bin/emacs", "/etc/fstab", (char *) 0 );
     }
  else
     {
       printf("You should see an editor window...\n");
     }
}
```



# Practise: Programmer (3)

Waiting for (termination of) child process: **`wait ()`**

```
#include <unistd.h>         /* sleep()                          */

main()
{
  int pid=fork();           /* Create child process             */
  if (pid == 0)
     {
       sleep(2);            /* go to sleep for 2 seconds        */
       printf("I'm the child. My PID is  %d\n", getpid() );
     }
  else
     {
       printf("I'm the parent. My child has PID  %d\n", pid);
       wait();              /* wait for child */
     }
}
```

# Practise: Programmer (4)

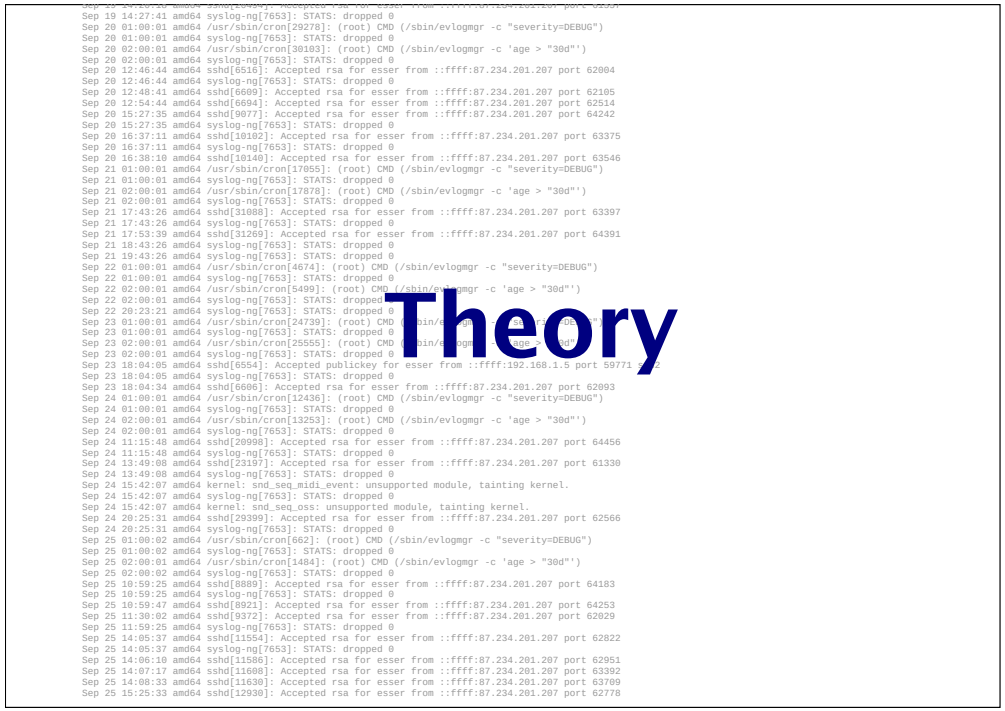
Truly more than one process:

After fork () two processes in process list

```
> pstree | grep simple
   |      |      |-bash---simplefork---simplefork

> ps w | grep simple
25684 pts/16 S+       0:00 ./simplefork
25685 pts/16 S+       0:00 ./simplefork
```





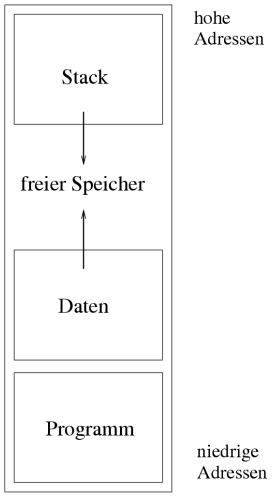

# Processes (1)

**Process details:**

- unique address space
- executable program (code, text)
- data (variables)
- program counter (instruction counter)
- stack and stack pointer
- contents of hardware registers (process context)



# Processes (2)




- Data: dynamically generated
- Stack: administers function calls
- Details: see chapter on memory management
- Stack and data „grow towards another“

# Processes (3)

## States

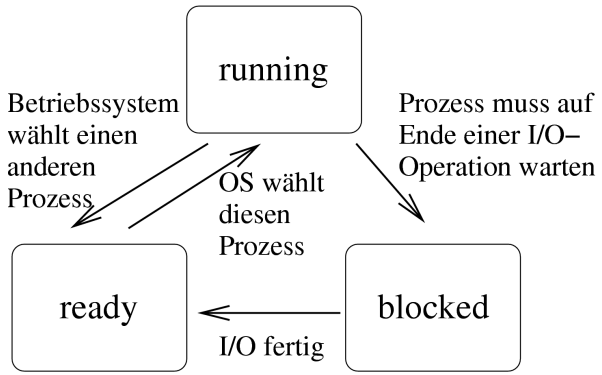
- **running:** active right now
- **ready:** would like to run
- **blocked / waiting:** waits for I/O

- **suspended:** ... by the user
- **sleeping:** waits for signal (IPC)
- **swapped:** data not in main memory (RAM)



# Processes (4)

## State transitions






# Processes (5)

## Process lists

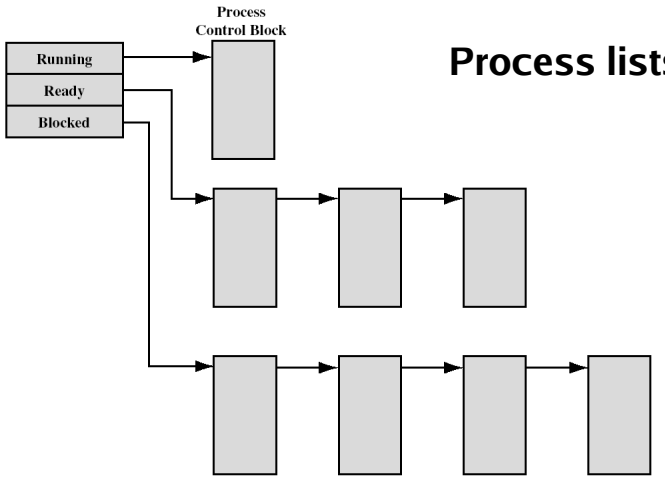



Bild: W. Stallings, S. 133



# Processes (6)

## Hierarchies

- processes create one another
- Creator is called parent process, the other one child process
- Children are autonomous (i. e.: have their own address space etc.)
- After process termination: return value goes to parent process

# Threads (1)

**What is a thread?**

- Thread of activity in a process
- one of several
- shared access to process data
- but: program counter, stack, stack pointer, hardware registers separate to each thread
- process scheduler handles threads – or doesn't (kernel vs. user level threads)



# Threads (2)

**Why Threads?**

- multi processor system: several threads can be active truly simultaneously
- If one thread is blocked by I/O, the others can continue working
- If a program logically consists of parallel tasks, programming with threads is simpler



# Threads (3): Examples

**Two different threads of activity:**
**Complex calculations with user requests**

Withoud threads:

```
while (1) {
  calculate_a_bit ();
  if user_input (x) {
    process_input (x)
  }
```



# Threads (4): Examples

**Complex calculations with user requests**

With threads:

```
T1:

while (1) {
  calculate_all ();
}
```

```
T2:

while(1) {
  if user_input (x) {
     process_input (x);
   }
 }
```

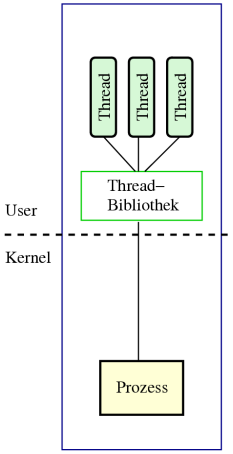
# Threads (5): Examples

**Server process handling lots of requests**

- process opens port
- For each incoming connection: Create new thread which deals with this request
- After termination of connection: destroy thread
- Advantage: No need for process creation (by the operating system!)



# User Level Threads

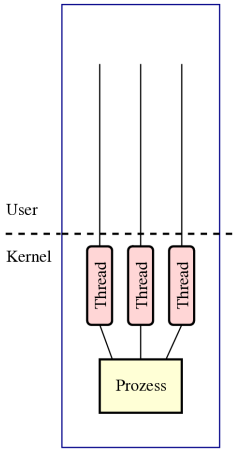



User Level Threads

- OS has no thread concept, only deals with processes
- program uses thread library which is responsible for:
  - creation, destruction
  - scheduling
- If a thread waits for I/O, the whole process waits
- otherwise pretty efficient



# Kernel Level Threads

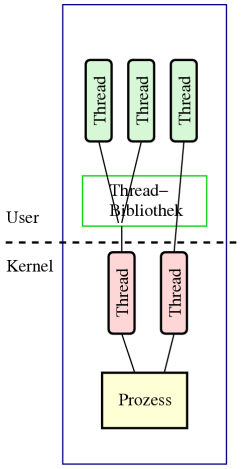



Kernel Level Threads

- OS knows threads
- OS handles threads:
  - creation, destruction
  - scheduling
- I/O of one thread won't block the others
- Time consuming: context switch between threads as complex as that between processes



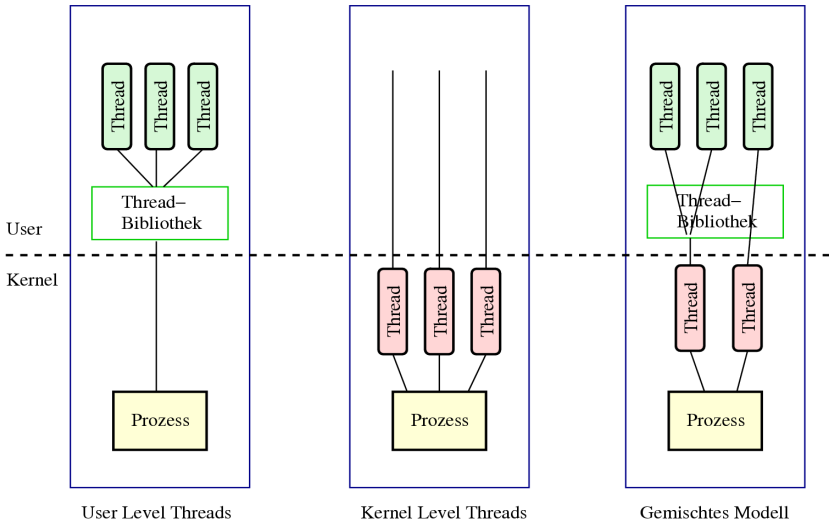
# Mixed Threads




Gemischtes Modell

- Combine both kinds of threads
- KL threads + UL threads
- thread library distributes UL threads onto KL threads
- e.g. I/O parts in one KL thread
- Best of both worlds:
  - I/O blocks only one KL thread
  - switching between UL threads is efficient
- SMP: use several CPUs

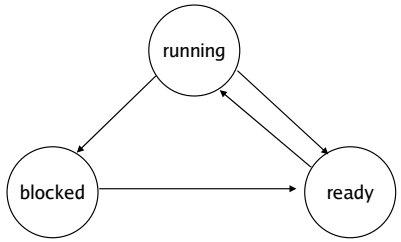
# Thread types, overview






# Thread states

- Process states suspended, sleeping, swapped etc. not used for threads
- Only three thread states








# Practice II: Threads



# Threads in C programs (1)

Linux: `pthread` library (POSIX Threads)

| | thread | process |
|---|---|---|
| Create | `pthread_create()` | `fork()` |
| Await termination | `pthread_join()` | `wait()` |

- include library headers
  `#include <pthread.h>`
- compile:
  `gcc -lpthread -o prog prog.c`

# Threads in C programs (2)

- Process: `fork()` creates identical copy / clone of calling process.
- Thread: `pthread_create()` gets function to be executed in new thread as argument.
- Thread: `pthread_join()` waits for a *specific* thread.



# Threads in C programs (3)

1. define thread function

```
void *thread_function(void *arg) {
  ...
  return ...;
}
```

2. create thread:

```
pthread_t thread;

if ( pthread_create( &thread, NULL,
     thread_function, NULL) ) {
    printf("Error while creating thread.\n");
    abort();
}
```



# Threads in C programs (4)

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

void *thread_function1(void *arg) {
  int i;
  for ( i=0; i<10; i++ ) {
    printf("Thread 1 says Hi!\n");
    sleep(1);
  }
  return NULL;
}

void *thread_function2(void *arg) {
  int i;
  for ( i=0; i<10; i++ ) {
    printf("Thread 2 says Hallo!\n");
    sleep(1);
  }
  return NULL;
}

int main(void) {

  pthread_t mythread1;
  pthread_t mythread2;

  if ( pthread_create( &mythread1, NULL,
       thread_function1, NULL) ) {
    printf("Error creating thread.");
    abort();
  }

  sleep(5);

  if ( pthread_create( &mythread2, NULL,
       thread_function2, NULL) ) {
    printf("Error creating thread.");
    abort();
  }

  sleep(5);

  printf("still here...\n");

  if ( pthread_join ( mythread1, NULL ) ) {
    printf("Error while joining.");
    abort();
  }

  printf("Thread 1 is gone.\n");

  if ( pthread_join ( mythread2, NULL ) ) {
    printf("Error while joining .");
    abort();
  }

  printf("Thread 2 is gone.\n");

  exit(0);

}
```



# Threads in C programs (5)

**No „parent“ or „child threads“**

- POSIX threads have no concept of relatedness as processes do (parent / child process)
- Waiting for a thread requires thread variable: `pthread_join (thread, ..)`

# Threads in C programs (6)

Process with several threads:

- only one entry in process list
- status: „l“, multi-threaded
- Command `ps -eLf` gives thread information
  - NLWP: number of light weight processes
  - LWP: thread ID

```
> ps auxw | grep thread
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
esser    12022  0.0  0.0  17976   436 pts/15   Sl+  22:58   0:00 ./thread

> ps -eLf | grep thread
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
esser    12166  4031 12166  0    3 23:01 pts/15   00:00:00 ./thread1
esser    12166  4031 12167  0    3 23:01 pts/15   00:00:00 ./thread1
esser    12166  4031 12177  0    3 23:01 pts/15   00:00:00 ./thread1
```