Betriebssysteme 2

进程是一个程序执行的实例。

1. Prozess
   - Definition: Instanziierung eines Programms in einem **Ablauf** mit einem konkreten **Zustand**
   - Ablauf:
     - eine Abfolge von Instruktionen, die ausgeführt werde
     - ein / oder mehrere Threads
   - Zustand:
     - 与 Ablauf 互相作用影响
     - Register
     - Adressraum: Heap, Stack, Programmcode
     - Offene Dateien
   - vs. Programm
     - Prozess: **Dynamische** Instanz von Code und Daten
     - Programm: **Statischer** Code und statische Daten
     - 同一个 Programm 可以有多个 Prozessinstanzen
   - vs. Thread
     - Thread: Leichtgewichtiger Prozess
       - ein Ablauf ohne eigenen Adressraum aber gewisse eigenen Zustand
       - ein Prozess kann mehrere Threads enthalten

2. Virtualisierung der CPU
   - Ziel: Jeder Prozess soll den Eindruck haben, die CPU nur für sich zu haben
   - Ressourcen lassen sich zeitlich(CPU Time-sharing) und räumlich(Memory, disk) aufteilen
   - Eingeschränkte direkte Ausführung wgen **privilegierte Operationen**, **Termination**, **effizient Ressourcen Verwaltung**(z.B. bei I/O)

3. Privilegierte Operationen
   - Ziel: Sicherstellen dass User-Prozess nicht miteinander schaden können
   - Lösung: Unterstützung von Hardware(spezielles Statusbit)
     - User Mode: eingeschränkter Mode
     - Kernel Mode: uneingeschränkt
   - Beispiel:
     - Zugriff auf gesamten Hauptspeicher
     - Disk I/O
     - Besondere x86 Instruktionen

4. Trap
   - a type of software interrupt used by a program
   - to request a service from the operating system (OS)
   - or to handle an exception or error in a controlled manner
   - allow a user program to **transfer control to the kernel**

```
read():
    movl $6, %eax ; Syscall-Table Index
    int $64       ; Trap-Table Index
```

   - user-space applications cannot directly interact with hardware resources
   - kernel can access the user-space memory directly to fill a read buffer, which is a memory area used to temporarily store data that is read from an external device
   - How it works:

- Triggering the interrupt by `$int` in x86 architecture
  - `$0x80`: triggers a Linux kernel system call(historically used in 32-bit machine)
  - `$64`: for the x86_64 architecture
- Switching to Kernel Mode
  - Saving the current program state(eg. register value, pc)
  - Jumping to the fixed address where the interrupt handler is located
  - Handling the interrupt: interrupt ID will be used to decide the concrete operations
  - Returning to user mode: the kernel restores program state, returns control to the program, resuming execution

5. System Call
   - Ziel: ein User-Prozess kann dadurch auf Geräte zugreifen
   - Definition: Systemaufruf, implementiert vom Betriebssystem
   - Funktion: ändert Privilege Level durch sog. Instruktion
6. Methodology: Wie CPU entziehen?
   - 操作系统如何分配 CPU 资源: **Mechanismus** und **Policy**
   - Mechanismus: Dispatching, um zwischen Prozessen zu wechseln
   - Policy: Scheduling Policy, um zu entscheiden, welcher Prozess wan ausgeführt wird
7. Dispatch-Mechanismus

```
while (1) {
    Prozess A für eine Zeitscheibe ausführen;
    Prozess A anhalten und seinen Kontext speichern;
    Kontext eines anderen Prozesses B laden;
}
```

   - Wie erlangt der Dispatcher die Kontrolle?
     - Kooperatives Multi-Tasking
       - Idee: Dem Prozess vertauen, dass er die CPU durch den Aufruf eines Traps abgibt
         - System Call
         - Page Fault: Seiten aufrufen, die nicht im Hauptspeicher Leichtgewichtiger
         - Fehler: Illegal Instruction, Divide by Zero
         - Spezieller System Call `yield()`
           - Idee: When a process calls `yield()`, it means it's done by using CPU for now, and the OS could switch to another process. It's not strict or `sleep` .
       - Nachteil: Prozess müssen sich selbst an die Regeln halten, falls ein Prozess alle Traps vermeidet und keine I/O ausführt, kann er die Maschine dauerhaft kontrollieren.
         - Einzige Lösung: Neustart!
       - Wird in modernen Betirebssystemen nicht verwendet

```c
void* task1(void* arg) {
    for (int i = 0; i < 5; i++) {
        printf("Task 1: Iteration %d\n", i);
        sched_yield();  // Yield the CPU to another thread
    }
return NULL;
}

void* task2(void* arg) {
    for (int i = 0; i < 5; i++) {
        printf("Task 2: Iteration %d\n", i);
```

```
            sched_yield();  // Yield the CPU to another thread
        }
    return NULL;
    }

    int main() {
        pthread_t thread1, thread2;

        // Create two threads that will run task1 and task2
        pthread_create(&thread1, NULL, task1, NULL);
        pthread_create(&thread2, NULL, task2, NULL);

        // Wait for both threads to finish
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);

        return 0;
    }
```

One example result is:

```
Task 2: Iteration 0
Task 2: Iteration 1
Task 2: Iteration 2
Task 1: Iteration 1
Task 1: Iteration 2
Task 1: Iteration 3
Task 1: Iteration 4
Task 2: Iteration 3
Task 2: Iteration 4
```

- Another option: Echtes Multi-Tasking
  - Idee: Das BS wird in regelmäßigen Zeitabständen automatisch betreten. Garantiert, dass das Betriebssystem regelmäßig die Kontrolle erlangt
    - Implementation: Die Hardware(CPU oder separater Chip) erzeugt einen Timer-Interrupt(z.B alle 10 ms)
  - 通过硬件触发的定时中断来强制操作系统介入。
  - 用户不可屏蔽定时器中断
  - 操作系统中的调度程序会根据定时中断的次数来决定何时进行进程切换。每当定时中断触发时，操作系统就会记录这一事件，直到达到设定的时间片为止。当达到一定数量的中断（即时间片到期）时，操作系统会进行 上下文切换，即暂停当前进程并调度下一个进程。
- Welcher Kontext muss gespeichert werden?
  - Der Dispatcher muss den Kontext des Prozesses verfolgen, wenn dieser nicht läuft
  - PCB: Kontext wird im Prozess Control Block gespeichert
    - PID: Prozess ID
    - Prozesszustand: z.B laufend, bereit oder blockiert
    - Ausführungszustand: Register, PC, Stack-Pointer
    - Scheduling-Priorität
    - **Hardware-Unterstützung**: PC und PSR wird beim Interrupts im Hardware gespeichert.
      - example

      ```
      Anwendung: Prozess A
      ---------
      Hardware: timer interrupt
      ```

```
Hardware: speichert regs(A)(User) in kernel-stack(A)
Hardware: zu Kernel-Mode wechseln
Hardware: springe zu Trap Handler
---------
BS: Trap behandeln
BS: switch() aufrufen
BS: speicherte regs(A)(Kernel) in proc-struct(A)
BS: lade regs(B)(Kernel) aus proc-struct(B)
BS: wechsele zu kernel-stack(B)
BS: return-from-trap (zu B)
---------
Hardware: lade regs(B)(User) von kernel-stack(B)
Hardware: zu User-Mode weckseln
Hardware: springe zu B's pc
---------
Prozess B
```

- Langsame I/O
  - z.B `read()`, `write()` greifen auf langsame Hardware zu
  - Zustand des Prozesses
    - Running: auf der CPU(nur einer auf einem Uniprozessor)
    - Ready: wartet auf die CPU
    - Blocked: wartet darauf, dass I/O oder Synchronisation abgeschlossen wird
  - BS
    - verfolgt jeden Prozess durch PID
    - verwaltet Warteschlangen(queues) für alle Prozesse
      - Ready-Queue: alle bereiten Prozesse
      - Event-Queue: eine logische Warteschlange pro Ereignis. Enthält alle Prozesse, die darauf warten, dass das Ereignis abgeschlossen wird.
- Prozesserstellung
  - Vollständig neuer Prozess
    - Schritte
      - ```
        1. spezifischen Code und Daten in den Speicher laden; Leeren Call-Stack
        erstellen
        2. PCB erstellen und initialisieren(so aussehen lassen wie einen
        Kontextwechsel)
        3. Prozess in die Ready-Queue einfügen
        ```
    - Vorteile: keine verschwendete Arbeit
    - Nachteile: Schwierig den Prozess korrekt einzurichten und alle möglichen Optionen abzubilden(Prozessberechtigungen, Umgebungsvariablen…)
  - Klonen und ändern
    - `fork()` : klont den aufrufenden Prozess
      - aktuellen Prozess anhalten und seinen Zustand speichern
      - Code, Daten, Stack kopieren
      - PCB erstellen und zur ready-Queue hinzufügen
    - `exec(char *file)` : überlagert aktuellen Prozess mit neuem Programm
      - ersetzt die aktuellen Daten- und Code-Segmente durch die in dem angegebenen Executable
    - Vorteile: flexibel, sauber, einfach
    - Nachteile: ineffizient, da erst eine Kopie erstellen und dann den Speicher überschreiben müssen
    - Process termination and return value:
      - `exit(int retval)` kann einen Prozess beenden
        ```

    - `retval` is the exit status of process. Check it with `echo $?`
    - ```
      int main() {
          return 0; // This is equal to exit(0)
      }
      ```
  - Kindprozess bleiben mit Elternprzess logisch verknüpft
  - `wait()`:
    - return value of the child process will be stored in its PCB
    - father process can get the return value of child by calling `wait()`
    - Zombie process: if the fahter doesn't call wait, then the Child's PCB still exists.
  - `sleep()`: hold the ongoing process for some seconds.

Example code:

1. fork.c

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
    puts("This is the beginning"); /* printed once */
    pid_t pid = fork();
    if (pid == -1) { /* in exceptional circumstances, fork can also fail */
        puts("Failed to create child.");
    } else if (pid == 0) { /* fork returns 0 to the child */
        puts("Hello from the child");
    } else { /* fork returns child's pid to the parent */
        printf("The child has pid: %i\n", pid);
    }
    puts("Goodbye!"); /* printed twice */
    return 0;
}
```

2. simpleshell.c

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

char *getcmd() {
    char *buf = malloc(256);
    scanf("%s", buf);
    return buf;
}

int main(int argc, char const *argv[])
{
    while (1) {
        char *cmd = getcmd();
        pid_t pid = fork();
        if (pid == 0) {
            // This is the child process. Setup the child's process environment here
            // E.g., where is standard I/O, how to handle signals?
            execl(cmd, cmd);
            // exec does not return if it succeeds
            printf("ERROR: Could not execute %s\n", cmd);
            exit(1);
        } else {
```
```

```
            // This is the parent process; Wait for child to finish
            // wait(NULL); // Simple wait call for only child
            printf("Child (PID: %d) running\n", pid);
            int status = 0;
            waitpid(pid, &status, 0);
            printf("Child exited with status %d\n", status);
        }
        free(cmd);
    }
    return 0;
}
```

3. zombie.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        puts("Hi, I'm the child, and I'll turn into a zombie now.");
        exit(42);
    } else {
        printf("Hi, I'm the parent, and the child has pid %d\n", pid);
        puts("Now is a good time to use ps to check for my zombie child!");
        sleep(30);
        int status = 0;
        waitpid(pid, &status, 0);
        printf("Child finished with exit code %d.\n", status);
        sleep(10);
        puts("Parent says goodbye!");
    }
    return 0;
}
```