Übungen Betriebssysteme (BS)

U1 – Prozesse verwalten

https://moodle.tu-dortmund.de/course/view.php?id=34604

Peter Ulbrich

peter.ulbrich@tu-dortmund.de

https://sys.cs.tu-dortmund.de/EN/People/ulbrich/

technische universität
dortmund

Theoriefrage 1

Erklärt den Unterschied zwischen fork(2) und vfork(2)? Gibt es zwischen den beiden heutzutage effektiv noch einen Unterschied?

Theoriefrage 1

Erklärt den Unterschied zwischen fork(2) und vfork(2)? Gibt es zwischen den beiden heutzutage effektiv noch einen Unterschied?

fork(2) erzeugt Kopie aus Vaterprozess

vfork(2) erzeugt neuen Prozess ohne Kopieren der Vaterprozess-Daten. Nützlich für direktes **exec(3)** Aufrufen. **vfork(2)** blockiert bis zum Aufruf von **exit(2)** oder **exec(3).**

Heutzutage effektiv copy-on-write bei fork & vfork

Theoriefrage 2

Erklärt zunächst, was die beiden Kommandos jeweils tun. Worin besteht nun der Unterschied?

ls -l > sort

ls -l | sort

Theoriefrage 2

Erklärt zunächst, was die beiden Kommandos jeweils tun. Worin besteht nun der Unterschied?

ls -l > sort

„ls -l “ listet den Inhalt des Verzeichnisses auf

“> sort” leitet die Ausgabe in die Datei „sort” um

ls -l | sort

Theoriefrage 2

Erklärt zunächst, was die beiden Kommandos jeweils tun. Worin besteht nun der Unterschied?

ls -l > sort

„ls -l “ listet den Inhalt des Verzeichnisses auf

“> sort” leitet die Ausgabe in die Datei „sort” um

ls -l | sort

„ls -l “ listet den Inhalt des Verzeichnisses auf

„| sort” leitet die Ausgabe von „ls -l” an das Programm „sort“ weiter

„sort“ gibt die Eingabe sortiert aus

Theoriefrage 3

Welchen für ein Betriebssystem wichtigen Zweck erfüllt der Systemaufruf wait(2)?

Theoriefrage 3

Welchen für ein Betriebssystem wichtigen Zweck erfüllt der Systemaufruf wait(2)?

Mittels **wait(2)** können beendete bzw. verwaiste Kindprozesse (Zombies) aufgeräumt werden.

Theoriefrage 4

```
for (;;) fork();
```

Was ist das Problem bei der Ausführung von diesem Code?

Theoriefrage 4

```
for (;;) fork();
```

Was ist das Problem bei der Ausführung von diesem Code?

- Jedes fork startet einen neuen Prozess
- Dem Betriebssystem gehen irgendwann die PIDs aus
  → Es können keine weiteren Prozesse gestartet werden
- Hohe CPU-Last, Speicherverbrauch etc.
- Umgangssprachlich: „fork-Bombe”
- Beschränkung durch **limits.conf(5)** möglich

Theoriefrage 4

```
for (;;) fork();
```

Verhalten in der 1. 2. 3. n. Generation?

Generation	1	2	3	n
Prozesse	1	2	4	2^{n-1}

Theoriefrage 4

```
for (;;) fork();
```

Verhalten in der 1. 2. 3. n. Generation?

Generation	1	2	3	n
Prozesse	1	2	4	2^{n-1}

→ Exponentielles Wachstum

Programmierung in C - a)

Einlesen von Standardeingabe, Anzeigen der Auswahl

```
#define NPARAMS 3

// Ausgabe der Parameter
void printApps(char const *params[]) {
    printf("Parameter:\n");
    for (int i = 0; i < NPARAMS; i++) {
        printf("  %d. %s\n", i, params[i]);
    }
}

int main() {
    // die Anwendungen definieren
    char const *params[NPARAMS];
    params[0] = "-a";
    params[1] = "-l";
    params[2] = "-t";

    printApps(params);
...
```

Programmierung in C - a)

Eingabe einlesen und auf Gültigkeit prüfen

```c
// Eingabe einlesen
int input;
printf("\nAuswahl: ");
int err = scanf("%d", &input);


// scanf-Rückgabewert prüfen
if (err < 0) {
    // Fehlerausgabe und abbrechen
} else if (err == 0) {
    // Ausgabe und weiter machen
}

// Eingabe auf Gültigkeit prüfen
if (input < 0 || input >= NPARAMS) {
    printf("Ungültige Eingabe: %d\n", input);
    continue;
}
```

Programmierung in C - b)

Starten von Programmen (ergänzt Code aus Teil a))

```c
// Auswahl in neuem Prozess ausführen
pid_t pid;
pid = fork();
if (pid == -1) {
    perror("Fehler bei fork");
    return EXIT_FAILURE;
} else if (pid > 0) { // Elternprozess
    // Status lesen und beenden
    int status;
    wait(&status); //Fehlerbeh. wegg.
    printf("PID von %s: %d\n\n", cmd, pid);
    return status;
} else { // Kindprozess
    execlp(cmd, cmd, params[input], NULL);
    perror("Fehler beim Ausführen des Programms");
    return EXIT_FAILURE;
}
```

Programmierung in C - c)

Wiederholung des Ablaufs (ergänzt Code aus Teil b))

- Schleife:
  alles in `while (1) { … }` einbetten
- Nach `scanf(…)`: Im Eingabepuffer verbleibende Zeichen entsorgen:

```
int c;
while ((c = getchar()) != '\n' && c != EOF);
```

- Eine weitere App namens exit aufnehmen, bei deren Auswahl Programm beenden:

```
// Programmende bei exit
if (input == 3) {
  return EXIT_SUCCESS;
}
```

Programmierung in C - Zusatz

Mit beliebigen Parametern starten (ergänzt Code aus Teil c))

- Beim Einlesen auf Buffergröße achten, sonst wie zuvor:

```
char custom[256];
err = scanf("%255s", custom)
```

- Für die Überprüfung der erfolgreichen Ausführung Code aus b) erweitern um eine Unterscheidung auf WIFEXITED(status).