Lernzettel

Interrupt Handling und Exception Handling in der Systemprogrammierung: Interrupt Service Routines, Fehlerbehandlung

Universität: Technische Universität Berlin
Kurs/Modul: Systemprogrammierung

Erstellungsdatum: September 6, 2025

Lernzettel: Interrupt Handling und Exception Handling in der Systemprogrammierung

(1) Grundbegriffe der Interrupt-Verarbeitung.

- Ein Interrupt ist ein Signal, das die laufende Programmausführung unterbricht und eine Interrupt-Service-Routine (ISR) aufruft.
- Eine Exception (Fehlerausnahme) ist ein spezieller Typ von Ereignis, oft durch Fehlerbedingungen ausgelöst (z. B. ungültiger Zustand, Division durch null).
- Hardware-Interrupts: z. B. IRQs, NMI (Non-Maskable Interrupt); Software-Interrupts werden durch Instruktionen ausgelöst (z. B. INT in x86).
- Interrupt-Controller oder IDT/IVT verwaltet Priorisierung, Maskierung und Zuordnung von Interrupts zu ISR-Adressen.
- Maskierung und Priorisierung: Interrupt-Enable/Disable, Prioritätsstufen, Interrupt-Latenz.
- Kontextwechsel: CPU speichert Kontext (Register, Programmzähler), Springen zur ISR, nach Beendigung Kontext wiederherstellen.

(2) Interrupt Service Routine (ISR).

- Ablauf eines Interrupts:
  - Interrupt tritt ein, CPU speichert Kontext (häufig automatisch durch Hardware).
  - Sprung zur zugeordneten ISR (Vektor-Tabelle/IDT).
  - ISR führt möglichst wenig Arbeit aus (schnell, deterministisch), ggf. Hinweis an Hardware (Acknowledge) und minutenschnelles Freigeben von Ressourcen.
  - Am Ende kehrt die CPU mit einem Return-from-Interrupt (IRET/eret) zurück zur ursprünglichen Ausführung.
  - Falls nötig: Interrupts innerhalb der ISR kurz wieder aktiviert, damit andere Interrupts verarbeitet werden können.
- Designprinzipien:
  - Minimaler Zeitaufwand in der ISR; schwere Arbeit in nachgelagerte Tasks verschieben (Bottom Half, Task-Queue, Deferred Work).
  - Verwende volatile Variablen für von ISRs genutzte gemeinsame Daten.
  - Vermeide Blockierungen, Deadlocks und lange Sperren.
- Top-Half vs. Bottom-Half:
  - Top-Half: schnelle Reaktion, z. B. Flag setzen, kurze Acknowledge-Routine.
  - Bottom-Half: aufwändige Verarbeitung in sicherem Kontext (z. B. Kernel-Thread, Workqueue).

(3) Fehlerbehandlung/Exception Handling in der Systemprogrammierung.

- Fehlercodes und errno: Funktionen geben Fehlerstatus zurück; errno liefert weitere Details.
- Propagation-Strategien: Fehler weiterreichen, auf höheren Ebenen behandeln, loggen.
- Exceptions in C: Da C keine native Exception-Mechanik hat, werden oft Setjmp/Longjmp als exception-like Muster eingesetzt.
- Schutzmechanismen: defensives Programmieren, Grenzchecks, Speicherschutz, Ressourcen-Backup und saubere Fehlerpfade.

(4) Beispiele in C (ISR-Skelett, Fehlerbehandlung).

```
// Beispiel 1: generisches ISR-Skelett (pseudo-C)
volatile int irq_pending = 0;

/* ISR für einen Timer-Interrupt (arch-abhängig) */
void isr_timer(void) {
  // Prolog: Kontext wird durch Hardware gesichert
  // Acknowledge Interrupt beim Controller
  irq_pending = 1;          // minimales Work: Signal an restliches System
  // ggf. Hintergrundarbeiten anmelden (Workqueue/Task)
  // Epilog: Return from Interrupt erfolgt durch die CPU/Hardware
}

// Beispiel 2: Bottom-Half-Ansatz (Pseudocode)
void timer_isr(void) {
  // minimal in ISR
  enqueue_work(process_timer_ticks); // Bottom-Half
}



// Beispiel 3: Fehlerbehandlung in C mit setjmp/longjmp (ungefähres Muster)
#include <setjmp.h>
#include <stdio.h>

static jmp_buf env;

void error_handler(void) {
  printf("Fehler erkannt, wechsle zu Fehlerpfad...\n");
  longjmp(env, 1);
}

int main(void) {
  if (setjmp(env) == 0) {
    // Normalpfad
    // ...
    // Bedingung, die zu Fehler führt
    error_handler();
  } else {
    // Fehlerpfad
```

```
    printf("Fehlerbehandlung abgeschlossen.\n");
  }
  return 0;
}
```

(5) Synchronisation und sichere Nutzung gemeinsamer Daten.

- Volatile-Qualifizierer für von ISR-Änderungen betroffene Variablen.
- Atomarität: Einsatz von Atomic-Operationen oder Sperren/Mutexen außerhalb von ISR (in sicherem Kontext).
- Speicherbarrieren und Konsistenzmodelle, um Sichtbarkeitsprobleme zu vermeiden.
- Nach dem ISR-Aufruf: klare Trennung von schnellem ISR-Teil und schwererer Verarbeitung außerhalb des Interrupt-Kontexts.

(6) Praxishinweise und Sicherheitsaspekte.

- Testen von Interrupt-Verhalten in simulierten Umgebungen und mit Hardware-Emulation.
- Minimierung der Per-Interrupt-Latenz; Priorisierung wichtiger Interrupts.
- Sichere Freigabe von Ressourcen, Vermeidung von Race Conditions.
- Sicherheit und Zuverlässigkeit: robuste Fehlerpfade, Logging, Wiederherstellung nach Fehlerzuständen.