Betriebssystemtechnik

Betriebsmittelzugriff

6./13. Juli 2009

Überblick

Motiv: Betriebsmittelvergabe

Koordinierter Zugriff auf wiederverwendbare/konsumierbare Betriebsmittel

Prozesse benötigen Betriebsmittel verschiedener Art und Anzahl, um weiter voranschreiten zu können

- statischer und dynamischer Arbeitsspeicher, persistenter Speicher
- Prozessor (CPU) und ggf. Koprozessor (FPU, GPU)
- Ein-/Ausgabegeräte
- sowie dazu korrespondierende Datenstrukturen der Software

Lernziel

- Notwendigkeit blockierender Synchronisation
- Vorbeugungsmaßnahme zur unkontrollierten Prioritätsumkehr
- blockadefreie Implementierung blockierender Systemfunktionen
- Belangtrennung bei Prozesssteuerung, -einplanung und -einlastung

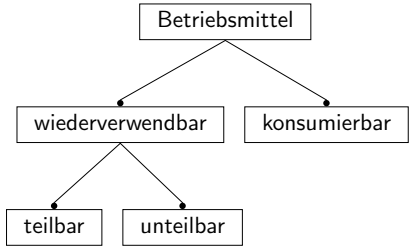
Einordnung

Schicht	Funktion	Konzepte
12	Programmverwaltung	Text, Daten, Überlagerung
11	Dateiverwaltung	Dateisystem; Verzeichnis, Verknüpfung
10	Prozessverwaltung	Aktivitätsträger, Kontext, Stapel
9	Adressraumverwaltung	Arbeitsspeicher, Segment, Seite
8	Informationsaustausch	Paket, Nachricht, Kanal, Portal
7	Geräteprogrammierung	Kern; Signal, Zeichen, Block, Datenstrom
6	Platzanweisung	Hauptspeicher, Fragment, Seitenrahmen
5	Zugriffskontrolle	Subjekt, Objekt, Domäne, Befähigung
4	Betriebsmittelzugriff	Verdrängungs-/Vorgangssperre
3	Auftragseinplanung	Ereignis, Priorität, Zeitscheibe, Energie
2	Ablaufsteuerung	Unterbrechungs-/Fortsetzungssperre, Wettlauftoleranz
1	Kontrollflusswechsel	Koroutine, Unterbrechung, Fortsetzung
0	Stammprozessorabstraktion	Stammsystem
−1	Peripherie	MMU, (A)PIC, DMA, UART, ATA, SCSI, USB, ...
−2	Zentraleinheit	ARM, AVR, PowerPC, SPARC, x86, ...

Rekapitulation: SOS 1 [1] bzw. SP [2], BS [3]

Betriebsmittel und Betriebsmittelarten

Wettbewerb um Betriebsmittel (engl. *resource contention*) bezieht sich auf Anzahl und Art eines Betriebsmittels



Betriebsmittelklassen

- Hardware
  - Speicher, Gerät
  - Prozessor(kern)
  - Signal (IRQ)
- Software
  - Puffer, Datei
  - Seite, Prozess
  - Signal, Nachricht

Beachte ↪ Zugriffssteuerung und Betriebsmittelart

einseitige Synchronisation ↦ konsumierbare Betriebsmittel

mehrseitige Synchronisation ↦ wiederverwendbare Betriebsmittel

Autorität von Betriebsmittelvergabe

Ausgewiesene oder (willkürlich) beliebige Instanz?

Betriebsmittel zugeteilt zu bekommen, um als Prozess effektiv und überhaupt Arbeit leisten zu können, ist eine Sache...

Zuteilung
- des Prozessors durch den Planer — *scheduler*
- des Busses durch den Schiedsrichter — *arbiter*
- der Kachel durch den Seitenwechsler — *pager*
- ⋮
- eines Datums bei Interprozesskommunikation

...ein zugeteiltes Betriebsmittel aber anderen Prozessen eigenmächtig vorzuenthalten, steht auf einem anderen Blatt

- Schutz kritischer Abschnitte durch Aussperrung von Prozessen
- letztlich ein Eingriff in die Autorität zentraler Zuteilungsfunktionen

Betriebsmittelart ⇝ Zugriffsart

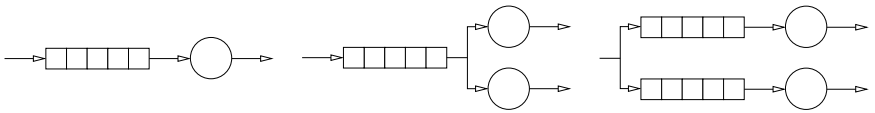
wiederverwendbar ⇌ in der Anzahl (physisch) begrenzt

teilbar
- unkoordinierter Zugriff
- uneingeschränkte Nebenläufigkeit
- ⇒ keine Synchronisation ✓

unteilbar
- koordinierter Zugriff
- eingeschränkte Nebenläufigkeit
- Kontrollflussabhängigkeit
- Wettstreit (*Reader*/*Writer*)
- ⇒ nichtblockierende Synchronisation ✓

konsumierbar ⇌ in der Anzahl (logisch) unbegrenzt
- koordinierter Zugriff
- eingeschränkte Nebenläufigkeit
- Datenflussabhängigkeit
- Kooperation (*Client*/*Server*)
- ⇒ blockierende Synchronisation ✗

Warteschlangen



- eine Warteliste
- ein Bediener
- Uniprozessor

- eine Warteliste
- zwei Bediener
- Multiprozessor
  - symmetrisch
- Verteilung
  - bei Freigabe

- zwei Wartelisten
- zwei Bediener
- Multiprozessor
  - asymmetrisch
- Verteilung
  - bei Ankunft

- eine Hierarchie von Wartelisten *und* Bediener wäre nicht unüblich

Warteschlangen (Forts.)

Bediener
- verarbeitende Komponente eines Wartesystems
  - Prozessor(kern), Gerät, Zusteller, ..., Prozess
- Zuteilung eines Auftrags bedeutet *Einlastung*
  - Mechanismus, abhängig von Bedienerart/-eigenschaften

Warteliste
- buchführende Komponente eines Wartesystems
  - Semaphor, Planer, Lagerhalter, ..., Gerätetreiber
- Aufnahme eines Auftrags geht mit *Einplanung* einher
  - Strategie, abhängig von Bedienerart/-eigenschaften

Beachte ↪ Dilemma
- eine gemeinsame Warteliste erhöht Durchsatz und Wettstreitigkeit
- getrennte Wartelisten senkt Durchsatz und Wettstreitigkeit

⇒ mehrstufige Organisation des Wartesystems:
- untergeordnete Warteliste(n) ▸ Verringerung von Wettstreitigkeit
- übergeordnete Warteliste(n) ▸ Erhöhung von Durchsatz

⇒ Hierarchie verwalteter Betriebsmittel (vgl. auch Kap. 3)

Austausch von Zeitsignalen

Zur Erinnerung: SOS 1 [1] bzw. SP [2]

Semaphor (engl. *semaphore*, [4])
- eine „nicht-negative ganze Zahl“
- für die zwei Elementarperationen definiert sind: P, V

P (hol. *prolaag*, „erniedrige“; auch *down*, *wait*)
- hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
- ansonsten wird der Semaphor um 1 dekrementiert

V (hol. *verhoog*, erhöhe; auch *up*, *signal*)
- inkrementiert den Semaphor um 1
- auf den Semaphor ggf. blockierte Prozesse werden deblockiert

Beachte ↪ Abstrakter Datentyp [5]
- zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen
  - deren Ausführung sich zeitlich überschneidet

Fallstudie: Semaphor

EWD — Edsger Wybe Dijkstra

```
#include "luxe/inline.h"
#include "luxe/tune/line.h"

typedef struct semaphore {
    line_t line;         /* optional waitlist: must be first member! */
    int load;
} semaphore_t;

extern void ewd_prolaag (semaphore_t *);
extern void ewd_verhoog (semaphore_t *);

INLINE void P (semaphore_t *this) { ewd_prolaag(this); }
INLINE void V (semaphore_t *this) { ewd_verhoog(this); }
```

```
#ifdef __fame_line_zilch
typedef struct {} line_t;
#endif
```

```
#ifdef __fame_line_chain
#include "luxe/chain.h"
typedef chain_t line_t;
#endif
```

```
#ifdef __fame_line_queue
#include "luxe/queue.h"
typedef queue_t line_t;
#endif
```


Fallstudie: Semaphor (Forts.)

nicht-negative ganze Zahl ▸ im logischen Sinn, Optimierungspotential:

$n \geq 0 \Rightarrow n$ = Anzahl erlaubter gleichzeitiger Prozesse

$n < 0 \Rightarrow |n|$ = Anzahl wartender (d.h., blockierter) Prozesse

```
void ewd_prolaag (semaphore_t *this) {
    if (this->load-- <= 0)
        sad_sleep(&this->line);
}
```

```
void ewd_prolaag (semaphore_t *this) {
    while (this->load-- <= 0) {
        sad_sleep(&this->line);
        this->load++;
    }
}
```

```
void ewd_verhoog (semaphore_t *this) {
    if (this->load++ < 0)
        sad_awake(&this->line);
}
```

```
void ewd_verhoog (semaphore_t *this) {
    if (this->load++ < 0)
        sad_flush(&this->line);
}
```

Hoare'sche Signalisierung
- *awake* wählt einen aus

Hansen'sche Signalisierung
- *flush* wählt alle aus

Fallstudie: Semaphor (Forts.)

Elementaroperation ▶ im logischen Sinn atomar, Synchronisierung:

konventionell ⇒ sperrend/blockierend ✓

unkonventionell ⇒ nichtsperrend/-blockierend ✗

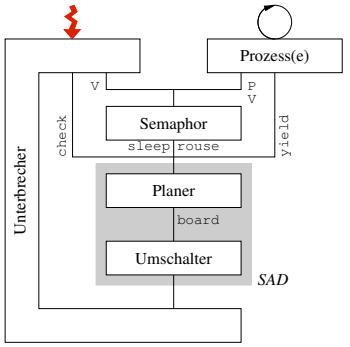
```
void ewd_prolaag (semaphore_t *this) {
    ENTER(ewd);
    ...
    LEAVE(ewd);
}
```

```
void ewd_verhoog (semaphore_t *this) {
    ENTER(ewd);
    ...
    LEAVE(ewd);
}
```

Schutzoptionen

nil	ungültig	
ice	Fortsetzungssperre	✓
irq	Unterbrechungssperre	✓
npx	verdrängungsfreier KA	✗
mux	gegenseitiger Ausschluss	

Infrastruktur zur Ein-/Umplanung: Funktionale Hierarchie



P/*V* ▶ *BM* anfordern/freigeben

sleep ▶ Freigabe von *BM* erwarten
▶ Prozessblockade/-auswahl

rouse ▶ Freigabe von *BM* anzeigen
▶ Modell *awake* oder *flush*

check ▶ Prozessorumplanung prüfen
▶ Zeitgeberunterbrechnung

yield ▶ Prozessen Vortritt gewähren

board ▶ Prozessorumschaltung

Beachte ↪ Leerlaufbetrieb (engl. *idle mode*)

- bei Prozessblockade findet die Auswahl keinen laufbereiten Prozess
- Leerlaufimplementierung durch Befehl oder Programm der Ebene 2
  - z.B. `hlt` (x86) oder aktives Warten auf Bereitlisteneinträge

Infrastruktur zur Ein-/Umplanung (Forts.)

SAD (Abk. für engl. *scheduling and dispatching*)

```
extern void sad_sleep (line_t *);   /* block thread on event */
extern void sad_awake (line_t *);   /* unblock one thread */
extern void sad_flush (line_t *);   /* unblock all threads */
extern void sad_rouse (line_t *);   /* awake or flush threads */

extern void sad_check ();           /* reschedule: asynchronous req. */
extern void sad_yield ();           /* reschedule: synchronous req. */

extern void sad_board (act_t *);    /* dispatch thread of control */
```

```
INLINE void sad_rouse (line_t *this) {
#ifdef __fame_sad_hoare
    sad_awake(this);
#else
    sad_flush(this);
#endif
}
```

Hoare oder Hansen?

- einfache Handhabung?
- robuste Implementierung?

↝ SOS 1 [1], SP [2]

Ausführungsstrang

Handlung, Vorgang (engl. *act*)

```
enum act_mood { ACT_NONPREEMPTIVE = 0x01, ACT_RELINQUISH = 0x2 };

typedef struct act {
    bid_t task;             /* scheduling state */
    enum act_mood mood;     /* coordination state */
    void *line;             /* blocked-on waitlist resp. signal */
    void *flux;             /* associated coroutine */
} act_t;

extern void   act_ready (act_t *);   /* ready to run, maybe preempt */
extern act_t *act_order ();          /* next ready to run act */

extern act_t *act_being ();          /* current act (on core) */
extern void   act_board (act_t *);   /* dispatch act (on core) */
```

- Spezialisierung eines Auftrags zur Prozessorvergabe: aktives Objekt
- Entkopplung von strategischen Maßnahmen der Prozesseinplanung

Einplanungseinheit

Angebot, Offerte (engl. *bid*) von bereitgestellten Aufträgen

```
enum bid_trim { BID_AVAILABLE, BID_READY, BID_RUNNING, BID_BLOCKED };

typedef struct bid {
    link_t link;          /* optional linkage: must come first! */
    enum bid_trim trim;   /* task state */
    int rank;             /* figure of merit */
} bid_t;

extern void   bid_ready (lot_t *, bid_t *);   /* add bid */
extern bid_t *bid_elect (lot_t *);            /* next bid, if any */
extern bid_t *bid_merit (bid_t *, bid_t *);   /* higher ranked bid */

extern lot_t *bid_hoard ();                   /* pool of bids */
```

- Abstraktion von Art und Erscheinung eines Auftrags: aktiv $\rightleftharpoons$ passiv
- Abstraktion von der Ausprägung einer Bedienstation: CPU/Gerät
- Kapselung rein strategischer Maßnahmen zur Auftragseinplanung

Prozesssteuerung: Bausteine

„schmerzlich“ (engl. *sad*)

Handlung (engl. *act*)

Angebot (engl. *bid*)

Lieferung (engl. *lot*)

Prozesssteuerung: Abstraktionen und Zuständigkeiten

SAD
- Ermöglichung wettlauftoleranter blockierender Funktionen
- Zustandsmaschine zur Steuerung von Ausführungssträngen

ACT
- Einplanung und Einlastung von Ausführungssträngen
- Auftragsspezialisierung für die Bedienstation „CPU“

BID
- Buchführung von Aufträgen an beliebige Bedienstationen
- Umsetzung der jeweiligen Einplanungstrategie

LOT
- Repräsentation und Verwaltung der Auftragsliste
- Abbildung auf statische oder dynamische Datenstrukturen

Beachte $\hookrightarrow$ Variantenvielfalt

SAD
- Arten der Signalisierung blockierter Ausführungsstränge

ACT
- verschiedene Gewichtsklassen von Ausführungssträngen

BID
- unterschiedliche Verfahren zur Auftragseinplanung

LOT
- fallspezifische Auslegung der Auftragsliste(n)

Verdrängungsfreie kritische Abschnitte

NPCS (Abk. für engl. *non-preemptive critical section*)
- Einplanung von Fäden läuft (nahezu) wie gewöhnlich durch
  - ausgelöste Fäden kommen auf die Bereitstellungsliste
  - der laufende Faden bekommt nicht den Prozessor entzogen
- nur die Einlastung von Fäden wird zeitweilig ausgesetzt
  - Verdrängungsereignisse werden zeitversetzt behandelt
  - d.h., fremd- wie auch selbstverursachte Verdrängungen[1]
- zeitweilige Monopolisierung des Prozessors durch einen Faden

Beachte $\hookrightarrow$ Determiniertheit & Verklemmung
- macht die Freigabe unteilbarer Betriebsmittel vorherseh-/sagbar
- macht die Nachforderung unteilbarer Betriebsmittel unteilbar

[1] Eine Verdrängung ist selbstverursacht, wenn der laufende Faden einen Faden höherer Priorität als er selbst bereitstellt. Sie ist fremdverursacht, wenn ein anderer (ggf. externer) Prozess den laufenden Faden dazu zwingt, den Prozessor abzugeben.

Analogie zur Fortsetzungssperre

SAD — minimale Erweiterung

```
extern void sad_avert ();          /* enter non-preemptive section */
extern void sad_admit ();          /* leave non-preemptive section */
extern void sad_waive ();          /* catch up preemption, if any */
```

Verdrängungsfreies *P*

```
void ewd_prolaag (semaphore_t *this) {
    sad_avert();
    if (this->load-- <= 0)
        sad_sleep(&this->line);
    sad_admit();
}
```

Verdrängungsfreies *V*

```
void ewd_verhoog (semaphore_t *this) {
    sad_avert();
    if (this->load++ < 0)
        sad_awake(&this->line);
    sad_admit();
}
```

Beachte ↪ Verdrängungsschutz ≠ Fortsetzungsschutz

- Verdrängungsschutz verzögert lediglich gleichzeitige Prozesse
- Fortsetzungen können nach wie vor zur Ausführung gelangen

⇒ ein *V* ausgelöst vom Unterbrecher kann *P* und *V* überlappen !!!

Verdrängungssteuerung

Verdrängung abwehren

```
void sad_avert () {
    act_t *self = act_being();              /* current thread */

    self->mood |= ACT_NONPREEMPTIVE;        /* processor sharing off */
}
```

Verdrängung zulassen

```
void sad_admit () {
    act_t *self = act_being();              /* current thread */

    self->mood &= ~ACT_NONPREEMPTIVE;       /* processor sharing on */
    if (self->mood & ACT_RELINQUISH)        /* preemption pending? */
        sad_waive();                        /* yes, catch up... */
}
```

- zwischenzeitig eintreffende Prozesse landen auf der Bereitliste
- *waive* lässt zurückgestellte Prozesse ggf. zu (vgl. TIP/ICE *clear*)

Verdrängungssteuerung (Forts.)

Verdrängung ersuchen

```
void sad_check () {
    act_t *self = act_being();              /* current thread */

    if (!(self->mood & ACT_NONPREEMPTIVE))  /* preemption enabled? */
        sad_waive();                        /* yes, reschedule */
    else self->mood |= ACT_RELINQUISH;      /* no, catch up later */
}
```

- verdrängbar zu sein oder nicht, wird als Prozessattribut aufgefasst
- Prozesse werden signalisiert, den Prozessor „freiwillig“ abzugeben

Beachte ↪ Bereitstellung von Prozessen

- seien *this* der bereitgestellte und *self* der aktuelle Prozess
- Verdrängung von *self* $\iff$ $PRIO(this) > PRIO(self)$
- bei Zurückstellung verbleibt *this* schlichtweg auf der Bereitliste

Einsatzbereich

NPCS schützt kritische Abschnitte, die mehr als ein wiederverwendbares unteilbares Betriebsmittel anfordern

- exklusive Belegung nur der CPU allein steht nicht im Vordergrund
- unterbrechungsbedingte Überlappungen des KA sind weiter möglich
- ebenso echte Parallelität: gesonderte Schutzverfahren sind gefordert

Eignung zeigt sich vielmehr in der Vorbeugung von (a) unkontrollierter Prioritätsumkehr und (b) Verklemmungen

(a)
- Betriebsmittel im KA blockiert von Prozess niedriger Priorität
- Zurückstellung unabhängiger Prozesse mittlerer Priorität

⇒ Wartezeitbegrenzung für abhängige Prozesse hoher Priorität [6]

(b)
- Nachforderung von Betriebsmitteln ist unteilbar

⇒ Entkräftung einer der vier Verklemmungsbedingungen [1, 2]