Systemprogrammierung

Prozesssynchronisation: Maschinenprogrammebene

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

11. November 2014

Prozesssynchronisation auf der Maschinenprogrammebene

Alleinstellungsmerkmal dieser Abstraktionsebene ist allgemein die durch ein **Betriebssystem** erreichte funktionale Anreicherung der CPU, hier:

(a) in Bezug auf die Einführung des Prozesskonzeptes und

(b) hinsichtlich der Art und Weise der Verarbeitung von Prozessen

Techniken zur Synchronisation gleichzeitiger Prozesse können demzufolge auf Konzepte zurückgreifen, die die Befehlssatzebene nicht bietet

zu (a) die Möglichkeit, Prozessinkarnationen kontrolliert schlafen legen und wieder aufwecken zu können

- Bedingungsvariable, Semaphor ✗
- *sleeping lock*

zu (b) die Möglichkeit, den Zeitpunkt der Einplanung oder Einlastung solcher Inkarnationen gezielt vorgeben zu können

- Verdrängungssperre ✗

Mehrseitige Synchronisation kritischer Abschnitte

Schutz kritischer Abschnitte durch **Ausschluss gleichzeitiger Prozesse** ist mit verschiedenen Ansätzen möglich

(a) asynchrone Programmunterbrechungen unterbinden, deren jeweilige Behandlung sonst einen gleichzeitigen Prozess impliziert

(b) Verdrängung des laufenden Prozesses aussetzen, die anderenfalls die Einlastung eines gleichzeitigen Prozesses bewirken könnte ✗

(c) gleichzeitige Prozesse allgemein zulassen, sie allerdings dazu bringen, die Entsperrung des KA eigenständig abzuwarten ✗

Alleingang (engl. *solo*) eines Prozesses durch einen kritischen Abschnitt sicherzustellen basiert dabei auf ein und dasselbe **Entwurfsmuster**:

```
CS_ENTER(solo);
:
CS_LEAVE(solo);
```

`CS_ENTER` (a) `cli`, (b) NPCS *enter*, (c) *P*, *lock*

`CS_LEAVE` (a) `sti`, (b) NPCS *leave*, (c) *V*, *unlock*

`solo` spezifiziert die fallabhängige Sperrvariable

Gliederung

Verdrängungsfreie kritische Abschnitte

NPCS, Abk. für (engl.) *non-preemptive critical section*

Ereignisse, die zur Verdrängung eines sich in einem kritischen Abschnitt befindlichen Prozesses führen könnten, werden unterbunden

enter
- Flagge zeigen, dass Entzug des Prozessors nicht stattfinden darf
- die mögliche Verdrängung des laufenden Prozesses zurückstellen

leave
- Flagge zeigen, dass Entzug des Prozessors stattfinden darf
- die ggf. zurückgestellte Verdrängungsanforderung weiterleiten

Aussetzen der Verdrängung des laufenden Prozesses ist durch (einfache) Maßnahmen an zwei Stellen der Prozessverwaltung möglich:

1. Einplanung eines freigestellten Prozesses zurückstellen
2. Einlastung eines zuvor eingeplanten Prozesses zurückstellen

Schutzvorrichtung (engl. *guard*): „Aufgaben durchschleusen“

Bitschalter (engl. *flag*) zum Sperren/Zurückstellen von Verdrängungen

Warteschlange zurückgestellter Verdrängungsanforderungen

Schutz eines kritischen Abschnitts vor Verdrängung

Abbildung auf die Zurückstellung von Prozeduraufrufen (vgl. S. 31)

```
ups_t npcs = {false, {0, &npcs.load.head}};

void npcs_enter()               { ups_avert(&npcs); }
void npcs_leave()               { ups_admit(&npcs); }
void npcs_check(order_t *task) { ups_check(&npcs, task); }
```

Geschützter KA

```
/*atomic*/ {
    npcs_enter();
    :
    npcs_leave();
}
```

Beispiel: Verdränger („*torpedo*“)

```
void __attribute__ ((interrupt)) torpedo() {
    npcs_check(&ps_check);
}
```

- ggf. zurückgestellte Prozesseinplanung (*ps_check*)

Beachte: Unabhängige gleichzeitige Prozesse

- werden unnötig zurückgehalten, obwohl sie den KA nicht durchlaufen

Gliederung

Bedingungsvariable (engl. *condition variable*)

Konzept für **bedingte kritische Abschnitte** [5], das zwei grundlegende Operationen definiert [4]:

await (auch: *wait*) $\models$ Unterbrechungssprotokoll ✗ $\rightsquigarrow$ S. 11

- lässt einen Prozess auf das mit einer Bedingungsvariablen assoziierten Ereignisses innerhalb eines KA warten:
  - gibt den gesperrten KA automatisch frei *und* blockiert den laufenden Prozess auf die Bedingungsvariable
  - bewirbt einen durch Ereignisanzeige deblockierten Prozess erneut um den Eintritt in den KA

cause (auch: *signal*) $\models$ Signalisierungsprotokoll ✗ $\rightsquigarrow$ S. 14

- zeigt das mit der Bedingungsvariable assoziierte Ereignis an
- deblockiert die ggf. auf das Ereignis wartenden Prozesse

Bedeutung

Ermöglicht einem Prozess, innerhalb eines kritischen Abschnitts zu warten, ohne diesen während der Wartephase belegt zu halten.

Bedingter kritischer Abschnitt

(engl.) *conditional critical section*, resp. *region*

Betreten des kritischen Abschnitts ist von einer Wartebedingung abhängig, die nicht erfüllt sein darf, um den Prozess fortzusetzen

- die Bedingung ist als **Prädikat** über die im kritischen Abschnitt enthaltenen bzw. verwendeten Daten definiert
- z.B. Fallunterscheidungen, Abbruchbedingungen (Schleifen)

Auswertung der Wartebedingung muss im kritischen Abschnitt erfolgen

- bei Nichterfüllung der Bedingung wird der Prozess auf Eintritt eines zur Wartebedingung korrespondierenden Ereignisses blockiert
  - damit das Ereignis später signalisiert werden kann, muss der kritische Abschnitt beim Schlafenlegen jedoch freigegeben werden
- bei (genauer: nach) Erfüllung/Signalisierung der Bedingung versucht der Prozess den kritischen Abschnitt wieder zu belegen
  - ggf. muss ein deblockierter Prozess die Bedingung neu auswerten

Systemorientierte Schnittstelle

Bedingungstyp

```
typedef struct condition {
    ...         // optional wait list
} condition_t;

void cv_await(condition_t *, solo_t *);
void cv_cause(condition_t *);
void cv_reset(condition_t *);
```

solo-Option: NPCS/UPS (S. 31)

```
typedef ups_t solo_t;

#define CS_ENTER(kind) ups_avert(kind)
#define CS_LEAVE(kind) ups_admit(kind)

#define CS_TAKEN(kind) ups_state(kind)
#define CS_CLEAR(kind) ups_treva(kind)
```

Datentyp mit optionaler Warteliste und assoziierter Sperrvariable

- die Sperrvariable (`solo_t*`) identifiziert einen kritischen Abschnitt
  - der in *cv_await* zunächst freigegeben und später wieder betreten wird
- die Liste enthält die durch die Wartebedingung blockierten Prozesse
  - wodurch *cv_cause* schnell den zu deblockierenden Prozess finden kann

Beachte: *cv_await* und *cv_cause* kontrollieren denselben KA

- beide müssen aus demselben gesperrten KA heraus aufgerufen werden
- einem „schläfrigen Prozess“ darf dabei das Wecksignal nicht entgehen

Entgangenes Wecksignal (engl. *lost wake-up*)

Prinzip — mit Problem(en)

```
void cv_await(condition_t *gate, solo_t *lock) {
    CS_LEAVE(lock);   // release critical section
    ps_sleep(gate);   // let process wait (asleep) on event
    CS_ENTER(lock);   // re-acquire critical section
}
```

Laufgefahr: Angenommen, der laufende Prozess hat den KA freigegeben (*CS_LEAVE* ausgeführt) und wird dann vor *ps_sleep* verdrängt:

1. Da der KA nun frei ist, kann das „Ereignis“ `gate` signalisiert werden, auf dessen Eintritt der Prozess mit *ps_sleep* passiv warten wollte.
2. Der Signalzustellung ist dieses Vorhaben des Prozesses jedoch nicht bekannt, so geht diesem dann das „Ereignis“ `gate` verloren.
3. Nach Wiedereinlastung wird sich der Prozess in *ps_sleep* blockieren und sodann ggf. vergebens auf den Ereigniseintritt warten.

Lösungsansatz: Abstraktion aufbrechen

Schlafenlegen: Eigentlich zu erwartende Implementierung

```
void ps_sleep(condition_t *gate) {
    ps_allot(gate);  // relate process to wait condition
    ps_block();      // finish CPU burst, reschedule CPU
}
```

Operation des Planers in zwei „elementaren“ Anweisungsschritten:

1. das Ereignis, auf dessen Eintritt sich der Prozess schlafen legen will, im Prozessdeskriptor verbuchen (*ps_allot*)
2. den Prozess blockieren, ihm dabei die CPU entziehen, die sodann einem laufbereiten Prozess zugeteilt wird (*ps_block*)

Herangehensweise zur Vorbeugung entgangener Wecksignale

1. das erwartete Ereignis dem Planer noch vor *CS_LEAVE* bekanntgeben
2. die Prozessblockierung dem Planer nach dem *CS_LEAVE* anzeigen

„Schläfrigen“ Prozess disponieren

Kritischen Abschnitt im „Halbschlaf“ verlassen (vgl. S. 34)

```
void cv_await(condition_t *gate, solo_t *lock) {
    cv_allot(gate);  // relate process to wait condition and wait list
    CS_LEAVE(lock);  // release critical section
    ps_block();      // let process wait (asleep)
    CS_ENTER(lock);  // re-acquire critical section
}
```

Lösungsansatz, der **besondere Vorsicht** im Planer erforderlich macht:

- nach *CS_LEAVE* kann die Fortsetzungsbedingung für einen noch laufenden Prozess signalisiert werden $\leadsto$ *cv_cause*
- der signalisierte Prozess kommt auf die Bereitliste, von der er sich durch *ps_block* ggf. selbst wieder entfernen und einlasten könnte

Analogie zum Sonderfall „Leerlauf“ (engl. *idle state*)

- holt ein sich schlafen legender Prozess sich selbst von der Bereitliste, bleibt er eingelastet und kehrt aus *ps_block* zurück

Fortsetzungsbedingung anzeigen

Wartebedingung aufheben

Ohne Warteliste

```
void cv_cause(condition_t *gate) {
    /* schedule blocked processes
     * awaiting the gate event */
    ps_rouse(gate);
}
```

Mit Warteliste

```
void cv_cause(condition_t *gate) {
    thread_t *next = cv_elect(gate);
    if (next)
        ps_ready(next);
}
```

Laufgefahr, sollte die Aufhebung der Wartebedingung nicht aus dem *cv_await* umfassenden kritischen Abschnitt heraus erfolgen:

- in dem Fall könnte *cv_cause* das „Ereignis“ `gate` überlappend mit der Ausführung des kritischen Abschnitts anzeigen
- kritisch ist der Teilabschnitt von Auswertung der Wartebedingung des KA bis Ausführung von *cv_await* bzw. (dem *ps_allot* in) *cv_allot*

- *cv_await* und *cv_cause* müssen paarweise dieselbe Bedingungsvariable (`gate`) bzw. denselben kritischen Abschnitt bedienen

Gliederung

Synchronisation durch Austausch von Zeitsignalen

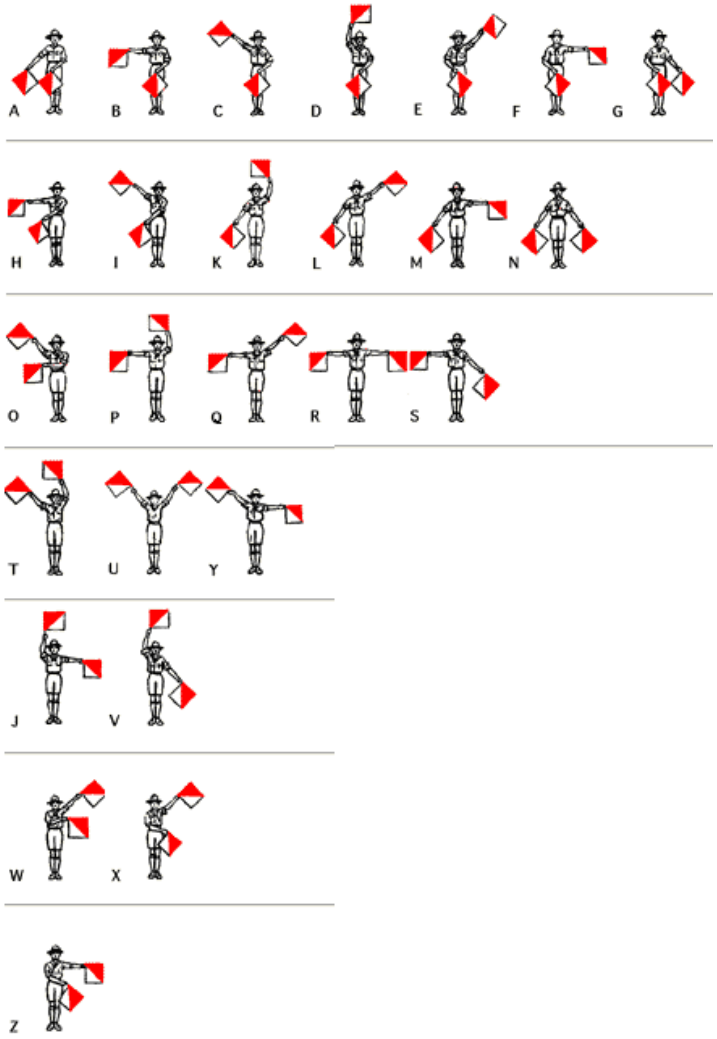
Semaphor (von gr. *sema* „Zeichen“ und *pherein* „tragen“) [1]

- eine „nicht-negative ganze Zahl“, für die — nach der ursprünglichen Definition [2] — zwei **unteilbare Operationen** definiert sind:
  P (hol. *prolaag* [1], „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* [1], erhöhe; auch *up*, *signal*)
  - inkrementiert den Semaphor um 1
  - auf den Semaphor ggf. blockierte Prozesse werden deblockiert
- ein abstrakter Datentyp zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen

[1] Allgemein ein Signalmast oder Winksignal, wie im Eisenbahnwesen bekannt.

Konzept zur Kommunikation und Koordination



EWD[2] beim Wort genommen

P

```
void ewd_prolaag(int *sema) {
    /*atomic*/ {
        if (*sema == 0) ps_sleep(sema);
        *sema -= 1;
    }
}
```

V

```
void ewd_verhoog(int *sema) {
    /*atomic*/ {
        *sema += 1;
        ps_rouse(sema);
    }
}
```

Programme für *P* und *V* bilden **kritische Abschnitte**:

1. gleichzeitiges Ausführen von *P* kann mehr Prozesse passieren lassen, als es der Semaphorwert (`sema`) erlaubt
2. gleichzeitiges Zählen kann Werte hinterlassen, die nicht der wirklichen Anzahl der ausgeführten Operationen (*P*, *V*) entsprechen
3. gleichzeitiges Auswerten der Bedingung (*P*) und Hochzählen (*V*) kann das Schlafenlegen (`ps_sleep()`) von Prozessen bedingen, obwohl die Wartebedingung für sie schon nicht mehr gilt („*lost wake-up*“)

[2] Edsgar Wybe Dijkstra

Konsequenzen für eine Implementierung von P und V

Pessimistischer Ansatz zum Schutz der kritischen Abschnitte, nämlich die **mehrseitige** (blockierende) **Synchronisation** von P und V

1. **wechselseitiger Ausschluss** wird die Funktionen bei ihrer Ausführung nicht überlappen lassen, weder sich selbst noch gegenseitig
   - P und V sind durch ein gemeinsames „Schloss“ zu schützen
2. Schlafenlegen eines Prozesses in P muss implizit die **Entsperrung** des kritischen Abschnitts zur Folge haben
   - sonst wird kein V die Ausführung vollenden können
   - als Folge werden in P schlafende Prozesse niemals aufgeweckt
3. Aufwecken von Prozessen in V sollte bedingt erfolgen, und zwar falls wenigstens ein Prozess in P schlafengelegt wurde

Opimistischer Ansatz als (bessere) Alternative, die den Schutz von P und V durch nichtblockierende Synchronisation erreicht

- äußerst knifflig, ein Thema für das fortgeschrittene Studium [7]

Mehrseitige Synchronisation von *P* und *V*

P

```
void wsp_prolaag(semaphore_t *sema) {
    CS_ENTER(&sema->lock);
    while (sema->load == 0)
        cv_await(&sema->gate, &sema->lock);
    sema->load--;
    CS_LEAVE(&sema->lock);
}
```

V

```
void wsp_verhoog(semaphore_t *sema) {
    CS_ENTER(&sema->lock);
    if (sema->load++ == 0)
        cv_cause(&sema->gate);
    CS_LEAVE(&sema->lock);
}
```

solo-Option: NPCS/UPS (S. 31)

- CS* Operationen vgl. S. 10

- vgl. THE-Semaphor auf S. 35

Kompositer Datentyp

```
typedef struct semaphore {
    int load;           // counter
    solo_t lock;        // safeguard
    condition_t gate;   // event
} semaphore_t;
```

Reflektion der Randbedingungen (S. 19) zur Implementierung von *P*/*V*:

zu 1. den wechselseitigen Ausschluss garantiert *CS_ENTER*

zu 2. die Entsperrung des KA im Wartezustand leistet *cv_await* (S. 13)

zu 3. bedingtes Aufwecken in *V*: $load = 0 \rightarrow$ mind. ein Prozess wartet

Arten von Semaphore

Instrumente zur Betriebsmittelvergabe, differenziert nach Wertebereichen der Semaphore

binärer Semaphor (engl. *binary semaphore*)

- verwaltet zu einem Zeitpunkt immer nur genau ein Betriebsmittel
  - wechselseitiger Ausschluss (engl. *mutual exclusion*, *mutex*)[3]
- vergibt **unteilbare Betriebsmittel** an Prozesse
- besitzt den Wertebereich $[0, 1]$

zählender Semaphor (engl. *counting semaphore*, *general semaphore*)

- verwaltet zu einem Zeitpunkt mehr als ein Betriebsmittel
  - d.h., mehrere Betriebsmittelexemplare desselben Typs
- vergibt teil- bzw. **konsumierbare Betriebsmittel** an Prozesse
- besitzt den Wertebereich $[0, N]$, für N Betriebsmittel

[3] „Mutex“ steht i.A. für einen binären Semaphor, mit dem zeitweilige Eigentümerschaft eines Fadens verknüpft ist: Nur für den Faden, der Mutex M besitzt, d.h., dem also $P(M)$ gelungen ist, wird $V(M)$ gelingen. (vgl. S. 36)

Arten von Betriebsmitteln

Semaphore und Betriebsmittelverwaltung

wiederverwendbare Betriebsmittel werden angefordert und freigegeben

- ihre Anzahl ist begrenzt: Prozessoren, Geräte, Speicher (Puffer)

  teilbar zu einer Zeit von mehreren Prozessen belegbar

  unteilbar zu einer Zeit von einem Prozess belegbar
- auch ein kritischer Abschnitt ist solch ein Betriebsmittel
  - von jedem Typ gibt es jedoch nur ein einziges Exemplar

konsumierbare Betriebsmittel werden erzeugt und zerstört

- ihre Anzahl ist (log.) unbegrenzt: Signale, Nachrichten, Interrupts

  Produzent kann beliebig viele davon erzeugen

  Konsument zerstört sie wieder bei Inanspruchnahme
- Produzent und Konsument sind voneinander abhängig

Ausschließender Semaphor

Vergabe unteilbarer Betriebsmittel, Schutz kritischer Abschnitte

```
semaphore_t lock = {1, 0};

int fai (int *ref) {
    int aux;

    P(&lock);
    aux = (*ref)++;
    V(&lock);

    return aux;
}
```

unteilbares Betriebsmittel „KA“

- von dem es nur ein Exemplar gibt
- der Initialwert des Semaphors ist 1

mehrseitige Synchronisation des KA

- die Reihenfolge gleichzeitiger Prozesse ist unbestimmt
- gleichzeitig können jedoch nicht mehrere Prozesse im KA sein

Syntaktischer Zucker

```
#define P(sema) wsp_prolaag(sema)
#define V(sema) wsp_verhoog(sema)
```

Signalisierender Semaphor

Vergabe konsumierbarer Betriebsmittel

```
char data;

semaphore_t full = {0, 0};

char consumer() {
    P(&full);
    return data;
}

void producer(char item) {
    data = item;
    V(&full);
}
```

konsumierbares Betriebsmittel

- ist vor dem Verbrauch zu erzeugen
- der Initialwert des Semaphors ist 0

einseitige Synchronisation

- nur einer von beiden beteiligten Prozessen wird ggf. blockieren
- nämlich der Konsument, wenn noch kein Datum verfügbar ist
- er ist später von dem Konsumenten wieder freizustellen

Begrenzter Datenpuffer: max. ein Platz

- Daten gehen verloren, wenn die Prozesse nicht im gleichen Takt arbeiten: $Konsument* \rightarrow (Produzent \rightarrow Konsument)+$

Semaphore „*considered harmful*“

Nicht alles „Gold“ glänzt...

- auf Semaphore basierende Lösungen sind komplex und fehleranfällig
  - Synchronisation: **Querschnittsbelang** nichtsequentieller Programme
    - kritische Abschnitte neigen dazu, mit ihren P/V-Operationen quer über die Software verstreut vorzuliegen
  - das Schützen gemeinsamer Variablen oder kritischer Abschnitte kann dabei leicht übersehen werden
- hohe Gefahr der **Verklemmung** (engl. *deadlock*) von Prozessen
  - umso zwingender sind Verfahren zur Vorbeugung, Vermeidung und/oder Erkennung solcher Verklemmungen
  - nichtblockierende Synchronisation ist mit diesem Problem nicht behaftet, dafür jedoch nicht immer durchgängig praktizierbar

- linguistische Unterstützung reduziert Fehlermöglichkeiten $\rightsquigarrow$ Monitor

Gliederung

Resümee

- Synchronisation in der Maschinenprogrammebene kann auf Konzepte von Betriebssystemen zurückgreifen
  - die den Zeitpunkt von Einplanung oder Einlastung gezielt beeinflussen
  - die Prozesse kontrolliert schlafen legen und wieder aufwecken
- durch eine **Verdrängungssperre** wird die Einplanung bzw. Einlastung von Prozessen erst verzögert wirksam
  - kritische Abschnitte werden verdrängungsfrei durchlaufen, aber
  - unabhängige gleichzeitige Prozesse werden unnötig zurückgehalten
- eine **Bedingungsvariable** ermöglicht Prozessen innerhalb eines KA zu warten, ohne diesen während der Wartephase belegt zu halten
  - ein Datentyp mit optionaler Warteliste und assoziierter Sperrvariable
  - *await* und *cause* müssen im selben gesperrten KA benutzt werden
- ein **Semaphor** ist ein kompositer Datentyp bestehend aus Zähl-, Sperr- und Bedingungsvariable
  - unterschieden wird zwischen binärem und zählendem Semaphor
  - ein *Mutex* ist ein binärer Semaphor mit Prozesseigentümerschaft

Literaturverzeichnis

[1] Dijkstra, E. W.:
Over seinpalen / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1964 ca. (EWD-74). –
Manuskript. –
(dt.) Über Signalmasten

[2] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

[3] Dijkstra, E. W.:
The Structure of the "THE"-Multiprogramming System.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346

[4] Hansen, P. B.:
Structured Multiprogramming.
In: *Communications of the ACM* 15 (1972), Jul., Nr. 7, S. 574–578

Literaturverzeichnis (Forts.)

[5] Hoare, C. A. R.:
Towards a Theory of Parallel Programming.
In: Hoare, C. A. R. (Hrsg.) ; Perrot, R. H. (Hrsg.): *Operating System Techniques*. New York, NY : Academic Press, Inc., Aug. – Sept. 1971 (Proceedings of a Seminar at Queen's University, Belfast, Northern Ireland), S. 61–71

[6] Parnas, D. L.:
Some Hypothesis About the "Uses" Hierarchy for Operating Systems / TH Darmstadt, Fachbereich Informatik.
1976 (BSI 76/1). –
Forschungsbericht

[7] Schröder-Preikschat, W. :
Betriebssystemtechnik.
http://www4.informatik.uni-erlangen.de/Lehre/SS??/V_BST/, jährlich. –
Vorlesungsfolien

Gliederung

Zurückstellung und Weiterleitung von Aufgaben

Universelle Schleuse (engl. *universal positing system*, UPS)

```
void ups_avert(ups_t *this) {
    this->busy = true;   // defer tasks
}
```

```
void ups_admit(ups_t *this) {
    ups_treva(this);     // let pass tasks
    if (ups_stock(this)) // any pending?
        ups_clear(this); // forward tasks
}
```

```
void ups_check(ups_t *this, order_t *task) {
    if (ups_state(this)) // defer?
        ups_defer(this, task);
    else                 // no, run task
        job_enact(&task->work);
}
```

```
typedef struct job job_t;

struct job {
    void (*call)(job_t*);
};
```

```
typedef struct order {
    chain_t next;
    job_t work;
} order_t;
```

```
typedef struct ups {
    bool busy;
    queue_t load;
} ups_t;
```

- **zurückgestellte Prozeduraufrufe** (engl. *deferred procedure calls*)

Prozesseinplanung (bedingt) zurückstellen

Seitenpfad heraus aus der Unterbrechungsbehandlung bewachen

Augenmerk ist auf jene Einplanungsfunktionen zu legen, die als Folge der Behandlung asynchroner Programmunterbrechungen aufzurufen sind

clock
- bei **Ablauf der Zeitscheibe** des laufenden Fadens

awake
- bei **Beendigung des E/A-Stoßes** eines wartenden Fadens

Beispiel: *clock*

```
extern void ps_clock();                    // scheduler's clock handler

order_t ps_order = {{0}, {ps_clock}};  // order to reschedule CPU

void __attribute__ ((interrupt)) clock() { npcs_check(&ps_order); }
```

- Zeitscheibenfunktion `ps_clock()` zur Prozessumplanung vorsehen
- Auftragsdeskriptor `ps_order` mit der Umplanungsfunktion aufsetzen und bei *Interrupt* mittels Steuerfunktion `npcs_check()` einspeisen

Prozesseinlastung zurückstellen

Übergang vom Prozessplaner zum Prozessabfertiger bewachen

Augenmerk ist auf die Einlastungsfunktion zu legen, die ggf. als Folge der Prozesseinplanung vom Planer aufgerufen wird

board

- zum **Umschalten des Prozessors** auf einen anderen Faden

Planer und Abfertiger lose koppeln (durch eine Art *lazy binding*)

- Umschaltfunktion `ps_board()` im Planer vorsehen
- diese mit Abfertigungsfunktion `pd_board()` assoziieren
- Brückenfunktion `pd_serve()` verbindet beide Einheiten
- Auftragsdeskriptor `pd_order` aufsetzen und mittels Steuerfunktion `npcs_check()` durchschleusen

```
extern void pd_board(thread_t *);

typedef struct board {
    order_t main;    // deferral request
    thread_t *plot;  // parameter placeholder
} board_t;

void pd_serve(job_t *task) {
    pd_board(((board_t*)task)->plot);
}

board_t pd_order = {{{0}, {pd_serve}}, 0};

void ps_board(thread_t *next) {
    pd_order.plot = next; // thread to be boarded
    npcs_check(&pd_order.main);
}
```

Unterbrechungsprotokoll: Prozess zumessen

Schläfrigen Prozess disponieren

```
void cv_allot(condition_t *gate) {
    cv_queue(gate, pd_being());  // queue process, possibly
    ps_allot(gate);              // relate process to wait condition
}
```

Schläfrigen Prozess bedingt auf die Warteliste setzen

```
void cv_queue(condition_t *gate, thread_t *this) {
#ifdef __fame_condition_queue
        ...                      // we have a queue: enter process
#endif
}
```

Gefahr von **Prioritätsverletzung** (engl. *priority violation*)

- die Einreihungsstrategie der Warteliste (Bedingungsvariable) muss konform gehen mit der Einreihungsstrategie der Bereitliste (Planer)
- *cv_queue* und *cv_elect* müssen (!) den Planer „benutzen“ [6]

Spezialisierter (zählender) Semaphor [3, S. 345]

Nichtpositiver Semaphorwert als ein Hinweis für die Anzahl wartender Prozesse

P

```
void wsp_prolaag(semaphore_t *sema) {
    CS_ENTER(&sema->lock);
    while (--sema->load < 0)
        cv_await(&sema->gate, &sema->lock);
    CS_LEAVE(&sema->lock);
}
```

Mit $load = 1$ zeigt V an, dass nur maximal ein Prozess den KA betreten darf.

V

```
void wsp_verhoog(semaphore_t *sema) {
    CS_ENTER(&sema->lock);
    if (sema->load >= 0)
        sema->load++;
    else {
        sema->load = 1;
        cv_cause(&sema->gate);
    }
    CS_LEAVE(&sema->lock);
}
```

Reflektion der Randbedingungen (S. 19) zur Implementierung von P/V:

zu 1. den wechselseitigen Ausschluss garantiert *CS_ENTER*

zu 2. die Entsperrung des KA im Wartezustand leistet *cv_await* (S. 13)

zu 3. bedingtes Aufwecken in V: $load < 0 \rightarrow |load|$ Prozesse warten

Spezialisierter (binärer) Semaphor: *Mutex*

Eigentümerschaft verbuchen (*P*) und überprüfen (*V*)

Datentyperweiterung

```
typedef struct mutex {
    semaphore_t sema;
    thread_t *link;    // owner
} mutex_t;
```

P bzw. *lock*

```
void xsp_prolaag(mutex_t *bolt) {
    CS_ENTER(&bolt->sema.lock);
    bolt->link = pd_being();
    ewd_prolaag(&bolt->sema);
    CS_LEAVE(&bolt->sema.lock);
}
```

V muss ggf. scheitern!

```
#undef NDEBUG
#include "luxe/assert.h"
```

V bzw. *unlock*

```
void xsp_verhoog(mutex_t *bolt) {
    assert(bolt->link == pd_being());

    CS_ENTER(&bolt->sema.lock);
    ewd_verhoog(&bolt->sema);
    bolt->link = 0;
    CS_LEAVE(&bolt->sema.lock);
}
```

Zusicherung (engl. *assertion*)

- ein Prozess gibt einen KA frei, den er überhaupt nicht belegt hatte ☹
- dies deutet auf einen schwerwiegenden Programmierfehler hin
- eine Fortsetzung der Programmausführung ist nicht mehr angebracht

Datenpuffer ohne Pufferbegrenzung: Ringpuffer

```
typedef struct ringbuffer {
    char data[NDATA];
    unsigned nput;    // write index
    unsigned nget;    // read index
} ringbuffer_t;

void rb_reset (ringbuffer_t *rb) {
    rb->nput = rb->nget = 0;
}

char rb_fetch (ringbuffer_t *rb) {
    return rb->data[rb->nget++ % NDATA];
}

void rb_store (ringbuffer_t *rb, char item) {
    rb->data[rb->nput++ % NDATA] = item;
}
```

Problemstellen

Füllstand log. Ablauf

- voll?
- leer?

füllen Zählen

- `nput++`

leeren Zählen

- `nget++`

Kritische Abschnitte

- *rb_fetch*
- *rb_store*

Puffersteuerung mittels Bedingungsvariable

Datenpuffer mit Pufferbegrenzung (engl. *bounded buffer*)

Datenpuffer begrenzter Speicherkapazität als **Ringpufferspezialisierung**:

```
typedef struct buffer {
    ringbuffer_t ring;
    unsigned int gage;
    solo_t lock;
    condition_t data;
    condition_t free;
} buffer_t;
```

```
void bb_reset (buffer_t *bb) {
    rb_reset(&bb->ring);
    bb->gage = NDATA;
    CS_CLEAR(&bb->lock);
    cv_reset(&bb->data);
    cv_reset(&bb->free);
}
```

`ring` Ringpufferspeicher

`gage` aktueller Pufferpegel

- Puffer ist initial leer
- `NDATA` freie Einträge

`lock` Sperrvariable

- KA ist initial offen

`data` Bedingungsvariable für *bb_fetch*

- $gage = NDATA \rightarrow cv\_await$

`free` Bedingungsvariable für *bb_store*

- $gage = 0 \rightarrow cv\_await$

Koordiniertes Leeren mittels Bedingungsvariable

```
char bb_fetch (buffer_t *bb) {
    char item;
    CS_ENTER(&bb->lock);
    while (bb->gage == NDATA)
        cv_await(&bb->data, &bb->lock);
    item = rb_fetch(&bb->ring);
    bb->gage += 1;
    cv_cause(&bb->free);
    CS_LEAVE(&bb->lock);
    return item;
}
```

Puffer leeren ist ein KA:

- wechselseitiger Ausschluss
- *bb_fetch* & *bb_store*

Wartebedingung:

- der Puffer ist leer

Fortsetzungsbedingung:

- Puffereintrag geleert

Entnahme eines Datums gibt ein wiederverwendbares Betriebsmittel frei

- die Anzahl der freien Puffereinträge erhöht sich um 1
- die Fortsetzungsbedingung zum Füllen kann signalisiert werden
- das Datum selbst ist ein konsumierbares Betriebsmittel

Koordiniertes Füllen mittels Bedingungsvariable

```
void bb_store (buffer_t *bb, char item) {
    CS_ENTER(&bb->lock);
    while (bb->gage == 0)
        cv_await(&bb->free, &bb->lock);
    rb_store(&bb->ring, item);
    bb->gage -= 1;
    cv_cause(&bb->data);
    CS_LEAVE(&bb->lock);
}
```

Puffer füllen ist ein KA:

- wechselseitiger Ausschluss
- *bb_store* & *bb_fetch*

Wartebedingung:

- der Puffer ist voll

Fortsetzungsbedingung:

- ein freier Puffereintrag konnte mit einem Datum belegt werden

Pufferung des Datums stellt ein konsumierbares Betriebsmittel bereit

- die Anzahl der freien Puffereinträge erniedrigt sich um 1
- die Fortsetzungsbedingung zum Leeren kann signalisiert werden
- der Puffereintrag selbst ist ein wiederverwendbares Betriebsmittel

Puffersteuerung mittels Semaphore

Bounded buffer revisited...

Ringpufferspezialisierung: „Dreiergespann“ von Semaphore...

```
typedef struct buffer {
    ringbuffer_t ring;
    semaphore_t lock;
    semaphore_t free;
    semaphore_t full;
} buffer_t;
```

```
void bb_reset (buffer_t *bb) {
    rb_reset(&bb->ring);
    wsp_initial(&bb->lock, 1);
    wsp_initial(&bb->free, NDATA);
    wsp_initial(&bb->full, 0);
}
```

`lock` sichert die Pufferoperationen

- wechselseitiger Ausschluss von lesen/schreiben

`free` verhindert Pufferüberlauf

- stoppt den Schreiber beim vollen Puffer

`full` verhindert Pufferunterlauf

- stoppt den Leser beim leeren Puffer

Koordiniertes Leeren mittels Semaphore

```
char bb_fetch (buffer_t *bb) {
    char item;
    P(&bb->full);
    P(&bb->lock);
    item = rb_fetch(&bb->ring);
    V(&bb->lock);
    V(&bb->free);
}
```

Szenario beim Leeren:

- einem leeren Puffer kann nichts entnommen werden
- freigewordener Pufferplatz soll wiederverwendbar sein
- Puffer leeren ist kritisch

einseitige Synchronisation $\mapsto$ zwei signalisierende Semaphore

- durch `full` ein konsumierbares Betriebsmittel anfordern
- durch `free` ein wiederverwendbares Betriebsmittel bereitstellen

mehrseitige Synchronisation $\mapsto$ ausschließender Semaphor `lock`

- sich selbst überlappendes Leeren und Leeren überlappendes Füllen

Koordiniertes Füllen mittels Semaphore

```
void bb_store (buffer_t *bb, char item) {
    P(&bb->free);
    P(&bb->lock);
    rb_store(&bb->ring, item);
    V(&bb->lock);
    V(&bb->full);
}
```

Szenario beim Füllen:

- voll ist voll...
- gepufferte Daten sollen konsumierbar sein
- Puffer füllen ist kritisch

einseitige Synchronisation $\mapsto$ zwei signalisierende Semaphore

- durch `free` ein wiederverwendbares Betriebsmittel anfordern
- durch `full` ein konsumierbares Betriebsmittel bereitstellen

mehrseitige Synchronisation $\mapsto$ ausschließender Semaphor `lock`

- sich selbst überlappendes Füllen und Füllen überlappendes Leeren

Laufgefährliches Leeren/Füllen mittels Semaphore

```
char bb_fetch (buffer_t *bb) {
    char item;
    P(&bb->lock);
    P(&bb->full);
    item = rb_fetch(&bb->ring);
    V(&bb->free);
    V(&bb->lock);
}
```

Was kann hier die Folge sein?

```
void bb_store (buffer_t *bb, char item) {
    P(&bb->lock);
    P(&bb->free);
    rb_store(&bb->ring, item);
    V(&bb->full);
    V(&bb->lock);
}
```

Verklemmungsgefahr

Angenommen, ein Prozess findet (a) beim Leeren, dass kein Datum *oder* (b) beim Füllen, dass kein freier Platz im Puffer verfügbar ist:

- Der Prozess wird dann im KA auf `full` oder `free` blockieren, den KA (`lock`) dabei aber nicht freigeben.
- Jeder andere Prozess, der ein Datum oder den freien Platz verfügbar machen könnte, würde dann beim Eintritt in diesen KA blockieren.