READERS & WRITERS

Gegeben sei eine globale Datei, auf die mehrere Prozesse lesend oder schreibend Zugriff haben sollen, und die konsistent gehalten werden soll.

Dieses Problem kommt in der Praxis häufig vor und ist ein gutes Beispiel zur Demonstration unterschiedlicher Synchronisationsmechanismen.

Anforderungen:

- Mehrere Leser dürfen parallel zugreifen.
- Während des Lesens ist kein Schreiben erlaubt.
- Während des Schreibens ist kein weiterer Lese- oder Schreibzugriff erlaubt.
- Wartende Schreiber haben Vorrang vor wartenden Lesern.

Zustandsvariablen:

```
rc   = Anzahl aktueller Leser
wc   = Anzahl aktueller und wartender Schreiber
busy = Schreibzugriff besteht
```

Wartelisten:

```
rl   = Liste der wartenden Leser
wl   = Liste der wartenden Schreiber
```

Illegale Zustände:

```
busy and (rc <> 0)
```

Unstabile Zustände:

```
(not busy) and (not empty(wl) or not empty(rl))
```

Vorgestellte Mechanismen:

- Semaphores (ad hoc-Lösung)
- Monitor
- Conditional Critical Regions
- Verwalterprozeß (naiv)
- Verwalterprozeß (sicher)
- es gibt noch weitere

Gemeinsame Annahmen:

- Es gibt zwei elementare Zugriffsfunktionen `do_read` und `do_write`, welche die eigentlichen Datentransporte ausführen, und die hier nicht näher spezifiziert sind. Einzelheiten der Zugriffe werden durch zusätzliche Parameter angegeben, die hier unterdrückt sind.
- Die Aufrufer verwenden zum Zugriff auf die Datei nur die Routinen `read` und `write`. Diese Routinen sind meistens lokal und haben dann im allgemeinen den folgenden Aufbau:

```
procedure read(params);
begin start_read;
      do_read(params);
      end_read;
end;
```

```
procedure write(params);
begin start_write;
      do_write(params);
      end_write;
end;
```

  Direkte Aufrufe von `do_read` und `do_write` führen im allgemeinen zu Inkonsistenzen und seien daher verboten.
- Die Routinen `start_read` etc. sind für die Synchronisation zuständig und je nach Lösungsmethode unterschiedlich.
- Es werden nicht immer alle oben angegebenen Variablen verwendet.
- Die Notation lehnt sich an Pascal an; die verwendeten Erweiterungen sollten ohne Erläuterung verständlich sein.

Lösung mit Semaphores (ad hoc)

nach Courtois, Heymans, Parnas 1971

```
module synch;
export start_read, start_write, end_read, end_write;

var rc, wc: integer;
    m, mr, mw, sr, sw: semaphore;

(* Hilfsroutinen: *)

procedure PP(var m, s: semaphore; var c: integer);
begin P(m);
      if c = 0 then P(s);
      c +:= 1;
      V(m);
end;

procedure VV(var m, s: semaphore; var c: integer);
begin P(m);
      c -:= 1;
      if c = 0 then V(s);
      V(m);
end;

(* Routinen zur Synchronisation: *)
```

```
procedure start_read;
begin P(m);
       P(sw);
        PP(mr, sr, rc);
       V(sw);
      V(m);
end;

procedure end_read;
begin VV(mr, sr, rc);
end;
```

```
procedure start_write;
begin PP(mw, sw, wc);
      P(sr);
end;

procedure end_write;
begin V(sr);
      VV(mw, sw, wc);
end;
```

```
begin (* Initialisierung *)
      rc := 0; wc := 0;
      V(sr); V(sw); V(mr); V(mw); V(m);
end; (* module synch *)
```

Bemerkung:

Diese Lösung ist speziell für dieses Problem konstruiert; sie zu verstehen, ist eine interessante Übungsaufgabe!

Lösung mit Monitor:

frei nach C.A.R.Hoare
und P.B.Hansen

```
monitor synch;
export start_read, end_read,
       write;

var rc, wc: integer;
    rl, wl: condition;


procedure start_read;
begin if wc <> 0 then delay(rl);
      rc +:= 1;
      continue(rl)
end;


procedure end_read;
begin rc -:= 1;
      if rc = 0 then continue(wl)
end;


procedure write(params);
begin wc +:= 1;
      if rc <> 0 then delay(wl);
      do_write(params);
      wc -:= 1;
      if wc <> 0 then continue(wl)
                 else continue(rl)
end;

begin (* Initialisierung *)
      rc := 0; wc := 0;
end; (* monitor synch *)
```

Bemerkung:

`write` ist hier direkt gegeben!

Die Lösung enthält einen Trick: überlegen Sie, was `continue(rl)` in `start_read` bewirkt!

Conditional Critical Regions:

nach P.B.Hansen: Operating System Principles

```
module synch;
export start_read, end_read,
       write;

var v: shared record
              rc, wc: integer;
       end;

procedure start_read;
begin region v do
      begin await wc = 0;
            rc +:= 1;
      end;
end;

procedure end_read;
begin region v do
      begin rc -:= 1;
      end;
end;

procedure write(params);
begin region v do
      begin wc +:= 1;
            await rc = 0;
            do_write(params);
            wc -:= 1;
      end;
end;

begin (* Initialisierung *)
      rc := 0; wc := 0;
end; (* module synch *)
```

Bemerkung:

`write` ist hier direkt gegeben!

Dies ist wohl der übersichtlichste Mechanismus, aber nicht leicht effizient zu implementieren.

Lösung mit Verwalterprozeß (naiv):

```
process synch;
var rc, wc: integer;
    busy: boolean;
    rl, wl: list of integer;
    idn: integer;
    buffer: (sr, er, sw, ew);

procedure init_read(idn: integer);
begin rc +:= 1;
      SEND_ANSWER(idn);
end;

procedure init_write(idn: integer);
begin busy := true;
      SEND_ANSWER(idn);
end;

(* Synchronisation und Prioritaet fuer Schreiber: *)

procedure start_read(idn: integer);
begin if wc = 0 then init_read(idn)
                else put_list(rl, idn);
end;

procedure end_read(idn: integer);
begin rc -:= 1;
      SEND_ANSWER(idn);
      if (rc = 0) and not empty(wl) then
      begin get_list(wl, idn); init_write(idn);
      end;
end;

procedure start_write(idn: integer);
begin wc +:= 1;
      if (rc = 0) and not busy then init_write(idn)
      else put_list(wl, idn);
end;

procedure end_write(idn: integer);
begin wc -:= 1; busy := false;
      SEND_ANSWER(idn);
      if not empty(wl) then
      begin get_list(wl, idn); init_write(idn);
      end else
      while not empty(rl) do
      begin get_list(rl, idn); init_read(idn);
      end;
end;
```

```
begin (* process body *)
      rc := 0; wc := 0;
      rl := []; wl := [];
      busy := false;
      repeat WAIT_ORDER(buffer, idn);
             case buffer of
                  sr: start_read(idn);
                  er: end_read(idn);
                  sw: start_write(idn);
                  ew: end_write(idn);
             end;
      until false;
end; (* process synch *)
```

Bemerkung:

Vorausgesetzt ist ein Nachrichtenmechanismus, der Aufträge (`SEND_ORDER`, `WAIT_ORDER`) und Antworten (`SEND_ANSWER`, `WAIT_ANSWER`) unterscheidet. Jeder Auftrag wird durch eine Antwort quittiert, die über ein beim Abholen des Auftrags ausgeliefertes Kennzeichen (`idn`) adressiert wird.

Die angegebene Lösung ist korrekt und flexibel, aber aus folgendem Grunde gefährlich:

Der Benutzer sollte normalerweise die folgenden Routinen verwenden:

```
procedure read;
begin SEND_ORDER(synch, sr);
      WAIT_ANSWER;
      do_read;
      SEND_ORDER(synch, er);
      WAIT_ANSWER;
end;
```

```
procedure write;
begin SEND_ORDER(synch, sw);
      WAIT_ANSWER;
      do_write;
      SEND_ORDER(synch, ew);
      WAIT_ANSWER;
end;
```

Der Benutzer kann aber auch `do_read` und `do_write` direkt aufrufen und so die Synchronisation umgehen!

Lösung mit Verwalterprozeß (sicher):

```
process synch;
var rc, wc: integer;
    busy: boolean;
    rl, wl: list of integer;
    idn: integer;
    buffer: record op: (rd, wrt);
                   nr: integer;
            end;
    msg: (order, answer);

procedure init_read(idn: integer);
begin rc +:= 1;
      SEND_ORDER(trans, (rd, idn));
end;

procedure init_write(idn: integer);
begin busy := true;
      SEND_ORDER(trans, (wrt, idn));
end;

procedure start_read(idn: integer);
begin if wc = 0 then init_read(idn)
                else put_list(rl, idn);
end;

procedure end_read(idn: integer);
begin rc -:= 1;
      SEND_ANSWER(idn);
      if (rc = 0) and not empty(wl) then
      begin get_list(wl, idn);
            init_write(idn);
      end;
end;

procedure start_write(idn: integer);
begin wc +:= 1;
      if (rc = 0) and not busy then init_write(idn)
      else put_list(wl, idn);
end;

procedure end_write(idn: integer);
begin wc -:= 1; busy := false;
      SEND_ANSWER(idn);
      if not empty(wl) then
      begin get_list(wl, idn); init_write(idn);
      end else
      while not empty(rl) do
      begin get_list(rl, idn); init_read(idn);
      end;
end;
```

```
begin (* process body *)
      rc := 0; wc := 0;
      rl := []; wl := [];
      busy := false;
      repeat WAIT_MESSAGE(msg, buffer, idn);
             if msg = order then
             case buffer.op of
                   rd: start_read(idn);
                  wrt: start_write(idn);
             end else
             case buffer.op of
                   rd: end_read(buffer.nr);
                  wrt: end_write(buffer.nr);
             end;
      until false;
end; (* process synch *)
```

Bemerkung:

Hier ist eine weitere Nachrichten-Operation `WAIT_MESSAGE` vorausgesetzt, die sowohl Aufträge wie Antworten ausliefert samt einer Angabe über die Art der Nachricht.

Die Routinen `start_read, start_write, end_read, end_write`, welche für die Synchronisations- und Prioritätsstrategie zuständig sind, sind mit denen in der naiven Lösung absolut identisch.

Die Auftraggeber verwenden hier die einfachen Routinen

```
procedure read;
begin SEND_ORDER(synch, rd);
      WAIT_ANSWER;
end;

procedure write;
begin SEND_ORDER(synch, wrt);
      WAIT_ANSWER;
end;
```

Jeder Auftraggeber richtet seine Bearbeitungswünsche an den Verwalterprozeß `synch`. Dieser erledigt die Synchronisation und gibt die Transportaufträge weiter an den Verwalterprozeß `trans`. Dessen Rückmeldungen werden zum Zweck der Synchronisation zur Kenntnis genommen und dann an den Auftraggeber weitergereicht.

Der Prozeß `trans` führt als einziger die Operationen `do_read` und `do_write` aus. Wenn er den ursprünglichen Auftraggebern nicht bekannt oder unzugänglich ist, oder wenn er nur Aufträge von `synch` annimmt, kann die Synchronisation nicht umgangen werden.

Wenn `trans` Besitzer der zu verwaltenden Datei ist, und wenn das Nachrichtensystem eindeutig und unverfälschbar eine Identifikation des Absenders einer Nachricht ausliefert, läßt sich auf die gleiche Weise eine Verwaltung von Zugriffsrechten organisieren.