Semaphore Ring Buffer

Shared Variables

var nrfull: semaphore := 0;
nempty: semaphore := N;
mutexP: semaphore := 1;
mutexC: semaphore := 1;
buffer: array[0..N−1] of message;
in, out: 0..N − 1 := 0, 0;

Producer i

loop

... Create a new message m;
—One producer at a time
P(mutexP);
—Await an empty cell
P(nempty);
buffer[in] := m;
in := (in + 1) mod N;
—Signal a full buffer
V(nrfull);
—Allow other producers
V(mutexP);
...
...
endloop

Consumer j

loop

... ...
—One consumer at a time
P(mutexC);
—Await a message
P(nrfull);

m := buffer[out];
out := (out + 1) mod N;
—Signal an empty buffer
V(nempty);
—Allow other consumers
V(mutexC);
Consume message m;
...
endloop
Weak Reader Preference Solution to the Reader-Writer Problem Using Semaphores

Shared Variables

var wmutex, rmutex: semaphore := 1, 1;
nreaders: integer := 0;

A Reader

loop
    —Readers enter one at a time
    P(rmutex);
    —First reader waits for reader’s turn,
    —then inhibits other writers
    if nreaders = 0 then
        P(wmutex)
    endif;
    nreaders := nreaders + 1;
    —Allow other reader entries/exits
    V(rmutex);
    Perform read operations;
    —Readers exit one at a time
    P(rmutex);
    nreaders := nreaders - 1;
    —Last reader allows writers
    if nreaders = 0 then
        V(wmutex)
    endif;
    —Allow reader entry/exit
    V(rmutex)
endloop

A Writer

loop
    —Each writer operates alone
    P(wmutex);
    Perform write operations;
    V(wmutex)
endloop
The Producer/Consumer Problem Using Eventcounts and Sequencers.

Shared Variables

var Pticket, Cticket: sequencer;
    In, Out: eventcount;
    buffer: array[0..N - 1] of message;

Producer i

— A variable, t, local
— to each producer
var t: integer;

loop

... Create a new message m;
— One producer at a time
t := ticket(Pticket);
await(In, t);
— Await an empty cell
await(Out, t - N + 1);
buffer[t mod N] := m;
— Signal a full buffer and
— allow other producers
advance(In);
... endloop

Consumer j

— A variable, u, local
— to each consumer
var u: integer;

loop

... Create a new message m;
— One consumer at a time
u := ticket(Cticket);
await(Out, u);
— Await a message
await(In, u + 1);
m := buffer[u mod N];
— Signal an empty buffer and
— allow other consumers
advance(Out);
Consume message m;
... endloop
A Weak Reader Preference Solution Using Sequencers and Eventcounts.

Shared Variables

\[ \text{var Wticket, Rticket: sequencer;} \]
\[ \text{Win, Rin: eventcount;} \]
\[ \text{nreade}rs: \text{integer} := 0; \]

A Reader

\[
\text{loop} \\
\ldots \\
\quad \text{—Readers enter one at a time} \\
\quad \text{await}(\text{Rin, ticket(Rticket)}); \\
\quad \text{—First reader waits for reader’s turn,} \\
\quad \text{—then inhibits other writers} \\
\quad \text{if nreade}rs = 0 \text{ then} \\
\quad \quad \text{await}(\text{Win, ticket(Wticket)}) \\
\quad \text{endif;} \\
\quad \text{nreade}rs := \text{nreade}rs + 1; \\
\quad \text{—Allow other reader entries} \\
\quad \text{advance(Rin);} \\
\quad \text{Perform read operations;} \\
\quad \text{—Readers exit one at a time} \\
\quad \text{await}(\text{Rin, ticket(Rticket)}); \\
\quad \text{nreade}rs := \text{nreade}rs - 1; \\
\quad \text{—Last reader allows writers} \\
\quad \text{if nreade}rs = 0 \text{ then} \\
\quad \quad \text{advance(Win)} \\
\quad \text{endif;} \\
\quad \text{—Allow reader entry/exit} \\
\quad \text{advance(Rin);} \\
\ldots \\
\text{endloop}
\]

A Writer

\[
\text{loop} \\
\ldots \\
\quad \text{—Each writer operates alone} \\
\quad \text{await}(\text{Win, ticket(Wticket)}); \\
\quad \text{Perform write operations;} \\
\quad \text{—Allow other writers (or} \\
\quad \text{—a reader) to lock out} \\
\quad \text{advance(Win);} \\
\ldots \\
\text{endloop}
\]
CIRCULARBUFFER: PROCEDURE OPTIONS (CONCURRENT);

CIRCULARBUFFERMONITOR: MONITOR;
DECLARE (BUFFERS (100)) CHARACTER (80) VARYING;
DECLARE (FIRSTBUFFER, LASTBUFFER) FIXED;
DECLARE (TOTALBUFFERS, FULLBUFFERS) FIXED;
DECLARE (ABUFFERISEMPTY) CONDITION;
DECLARE (ABUFFERISFULL) CONDITION;

DO;
    FIRSTBUFFER = 1;
    LASTBUFFER = 1;
    TOTALBUFFERS = 100;
    FULLBUFFERS = 0;
END;

SPOOLER: ENTRY (IMAGE);
DECLARE (IMAGE) CHARACTER (*) VARYING;
IF FULLBUFFERS = TOTALBUFFERS THEN
    WAIT (ABUFFERISEMPTY);
    BUFFERS (LASTBUFFER) = IMAGE;
    LASTBUFFER = MOD (LASTBUFFER, TOTALBUFFERS) + 1;
    FULLBUFFERS = FULLBUFFERS + 1;
    SIGNAL (ABUFFERISFULL);
END;

DESPoolER: ENTRY (IMAGE);
DECLARE (IMAGE) CHARACTER (*) VARYING;
IF FULLBUFFERS = 0 THEN
    WAIT (ABUFFERISFULL);
    IMAGE = BUFFERS (FIRSTBUFFER);
    FIRSTBUFFER = MOD (FIRSTBUFFER, TOTALBUFFERS) + 1;
    FULLBUFFERS = FULLBUFFERS - 1;
    SIGNAL (ABUFFERISEMPTY);
END;

READCARS: PROCESS;
DECLARE (CARDIMAGE) CHARACTER (80) VARYING;
CARDIMAGE = 'MORECARDS';
DO WHILE (CARDIMAGE <> 'ENDOFFILE');
    GET SKIP EDIT (CARDIMAGE) (A(80));
    CALL SPOOLER (CARDIMAGE);
END;

PRINTLINES: PROCESS;
DECLARE (LINEIMAGE) CHARACTER (80) VARYING;
LINEIMAGE = 'MORECARDS';
DO WHILE (LINEIMAGE <> 'ENDOFFILE');
    CALL DESPOOLER (LINEIMAGE);
    PUT SKIP EDIT (LINEIMAGE) (A(80));
END;

CSP/k program for managing a circular buffer.

readers_and_writers: monitor
  var readercount: integer;
  busy: boolean;
  OKtoread, OKtowrite: condition;

procedure startread;
  begin
    if busy then OKtoread.wait endif;
    readercount := readercount + 1;
    if OKtoread.queue then OKtoread.signal endif
  end startread;

procedure endread;
  begin
    readercount := readercount - 1;
    if readercount = 0 then OKtowrite.signal endif
  end endread;

procedure startwrite;
  begin
    if readercount ≠ 0 or busy then OKtowrite.wait endif;
    busy := true
  end startwrite;

procedure endwrite;
  begin
    busy := false;
    if OKtoread.queue then OKtoread.signal
    else OKtowrite.signal endif
  end endwrite;

begin
  readercount := 0; busy := false
end readers_and_writers;