Course 02158

Semaphores

Hans Henrik Løvengreen

DTU Compute

Semaphore Definition

- Dutch computer scientist
- Early 60'ies – solving concurrency problems in the THE Operating System

S : *semaphore*	**var** s : *integer* $:= 1$
`P`(S)	$\langle\, s > 0 \rightarrow s := s - 1 \,\rangle$
`V`(S)	$\langle\, s := s + 1 \,\rangle$

- Discovery: Can solve any synchronization problem (using auxiliary variables)
- 1965: Wrote an article about this and became (world) famous!

Critical Region with Semaphores

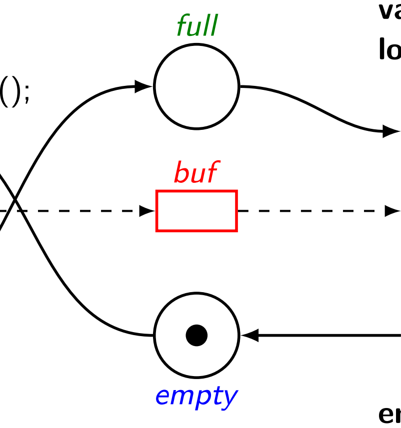
- **var** *mutex* : *semaphore* := 1;

  process $P[i : 1..n]$
    loop
      `P`(*mutex*);
      critical section$_i$;
      `V`(*mutex*);
      noncritical section$_i$;
    end loop

Producer/Consumer — single slot buffer

- **var** *buf* : T;
  full : *semaphore* := 0;
  empty : *semaphore* := 1;

process *Producer*
  var *data* : T;
  loop
    data := produce();
    P(*empty*);
    buf := *data*;
    V(*full*);
  end loop

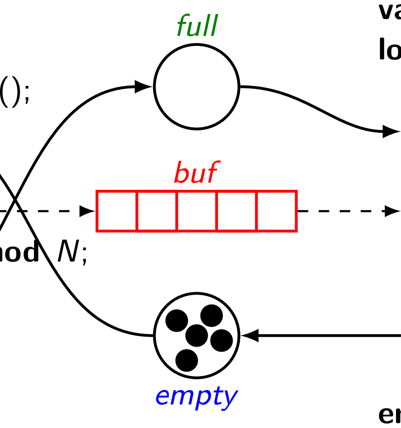


process *Consumer*
  var *result* : T;
  loop
    P(*full*);
    result := *buf*;
    V(*empty*);
    consume(*result*);
  end loop

Producer/Consumer — bounded buffer

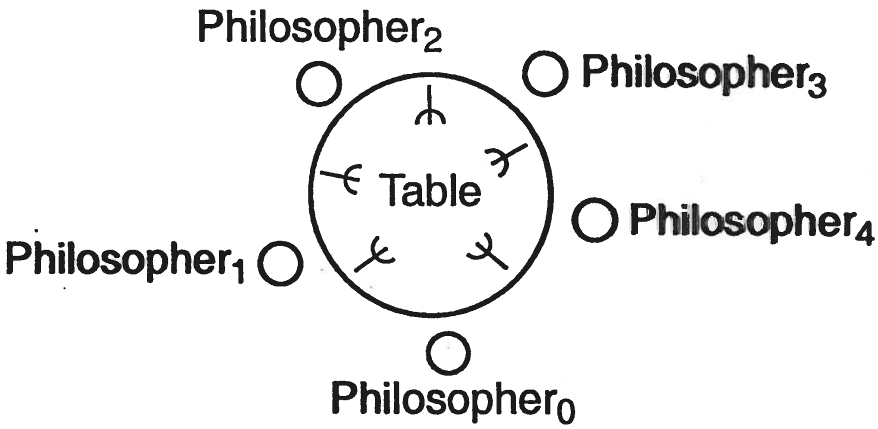
- **var** *buf*$[N]$: T; *in*, *out* : *integer* := 0
  full : *semaphore* := 0;
  empty : *semaphore* := N;
  $mutex_P$, $mutex_C$: *semaphore* := 1;

process *Producer*$[i : 1..n]$
  var *data* : T;
  loop
    data := produce();
    P(*empty*);
    P($mutex_P$)
    buf[*in*] := *data*;
    in := $(in + 1)$ **mod** N;
    V($mutex_P$)
    V(*full*);
  end loop

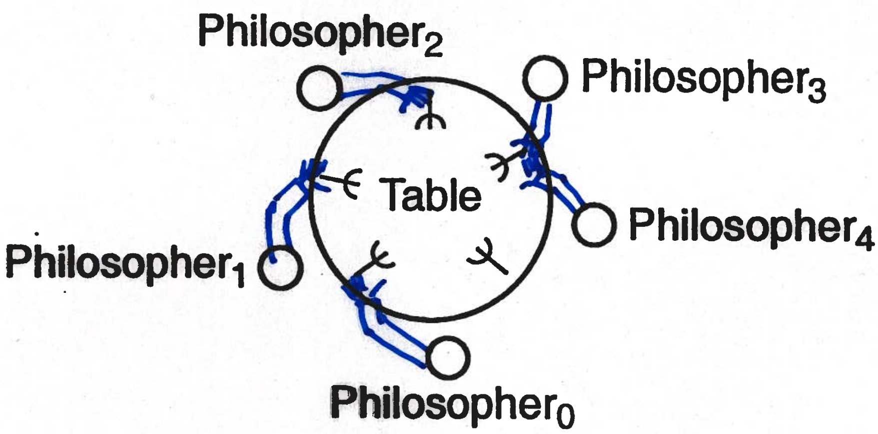


process *Consumer*$[j : 1..m]$
  var *result* : T;
  loop
    P(*full*);
    P($mutex_C$)
    result := *buf*[*out*];
    out := $(out + 1)$ **mod** N;
    V($mutex_C$)
    V(*empty*);
    consume(*result*);
  end loop

Dining Philosophers



Dining Philosophers

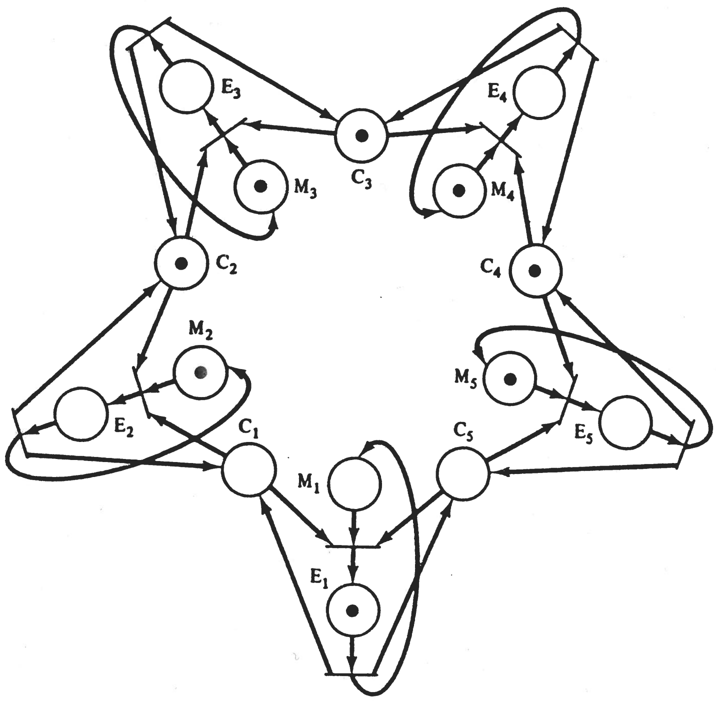


Dining Philosophers

```
sem fork[5] = {1, 1, 1, 1, 1};
process Philosopher[i = 0 to 3] {
  while (true) {
    P(fork[i]); P(fork[i+1]); # get left fork then right
    eat;
    V(fork[i]); V(fork[i+1]);
    think;
  }
}
process Philosopher[4] {
  while (true) {
    P(fork[0]); P(fork[4]);   # get right fork then left
    eat;
    V(fork[0]); V(fork[4]);
    think;
  }
}
```

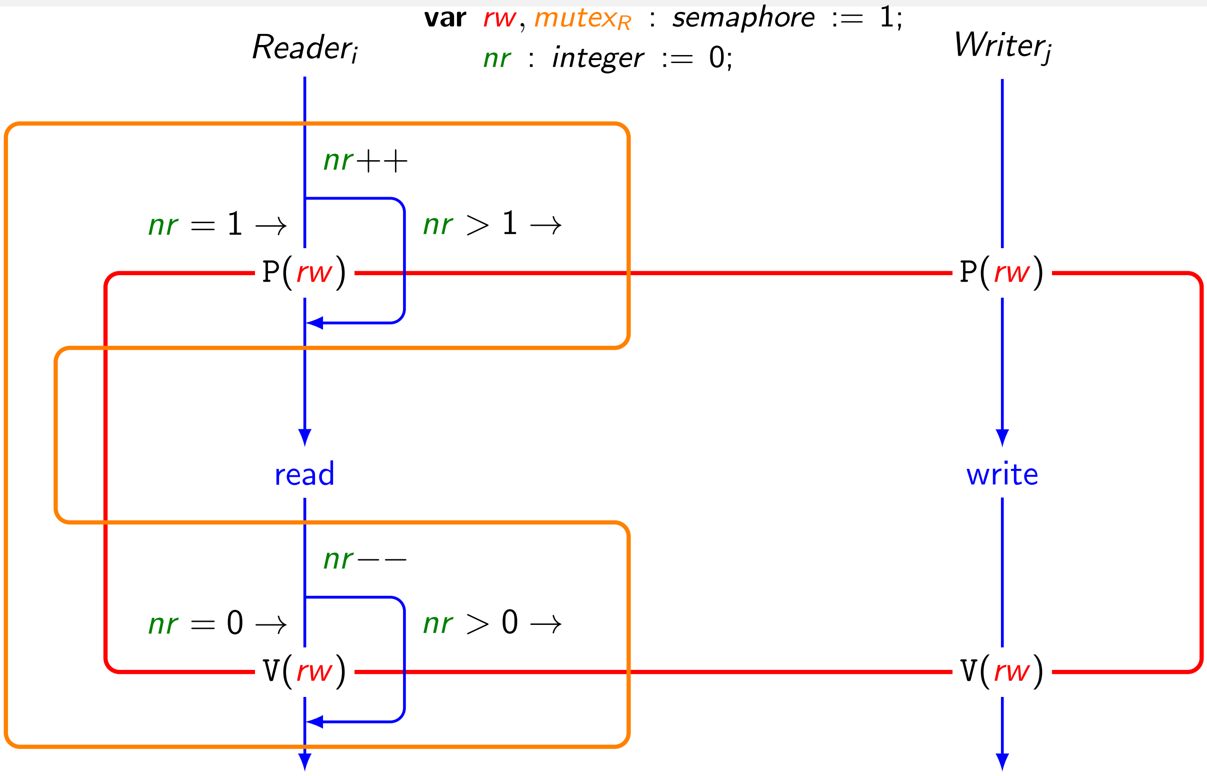
Figure 4.7 Dining philosophers solution using semaphores.

Dining Philosophers



Reader/Writer Synchronization

var $rw, mutex_R$: *semaphore* := 1;
nr : *integer* := 0;



Reader/Writer Synchronization — ad hoc

- **var** rw : *semaphore* := 1;
  nr : *integer* := 0;
  $mutex_R$: *semaphore* := 1;

process $Reader[i : 1..m]$
loop
…
P($mutex_R$);
$nr{+}{+}$;
if $nr = 1$ **then** P(rw);
V($mutex_R$);
read
P($mutex_R$);
$nr{-}{-}$;
if $nr = 0$ **then** V(rw);
V($mutex_R$);
…
end loop

process $Writer[i : 1..n]$
loop
⋮
P(rw);
write
V(rw);
⋮
end loop

Semaphore Properties — Safety

- Semaphore S represented by $s : integer := s_0$ $(s_0 \geq 0)$
- $\mathtt{P}(S)$: $\langle\, s > 0 \rightarrow s := s - 1;\ \#\mathtt{P}(S) := \#\mathtt{P}(S) + 1 \,\rangle$

  $\mathtt{V}(S)$: $\langle\, s := s + 1;\ \#\mathtt{V}(S) := \#\mathtt{V}(S) + 1 \,\rangle$

Semaphore invariant

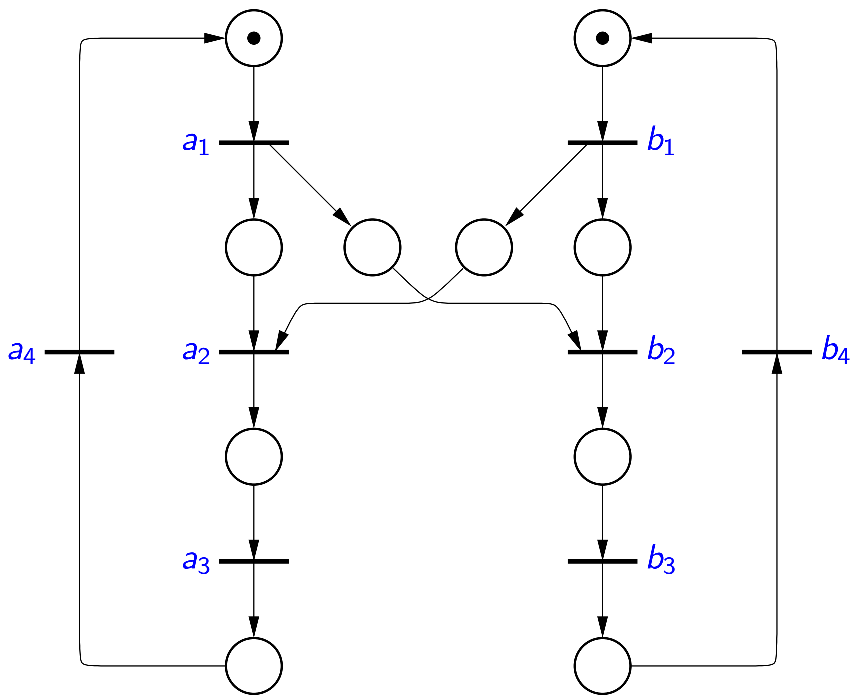
- Introduce $\#\mathtt{P}(S)$ and $\#\mathtt{V}(S)$ — initially 0
- Invariant

$$\begin{aligned} & s = s_0 + \#\mathtt{V}(S) - \#\mathtt{P}(S) \ \wedge \\ & s \geq 0 \end{aligned}$$

- Implies

$$\#\mathtt{P}(S) \leq s_0 + \#\mathtt{V}(S)$$

Lock-Step Synchronization



Lock-Step Proof

- **var** S_A, S_B : *semaphore* := 0;
  a, b : *integer* := 0;

process P_A
  repeat
    $\mathtt{V}(S_B)$;
    $\mathtt{P}(S_A)$;
    $op_A\, a{+}{+}$;
  forever

process P_B
  repeat
    $\mathtt{V}(S_A)$;
    $\mathtt{P}(S_B)$;
    $op_B\, b{+}{+}$;
  forever

- Show: $|a - b| \leq 1$
- By local arguments: $a \leq \#\mathtt{P}(S_A) \leq \#\mathtt{V}(S_B) \leq a + 1$

  $b \leq \#\mathtt{P}(S_B) \leq \#\mathtt{V}(S_A) \leq b + 1$

- Using: $\#\mathtt{P}(S_A) \leq \#\mathtt{V}(S_A)$: $a \leq b + 1$
  and: $\#\mathtt{P}(S_B) \leq \#\mathtt{V}(S_B)$: $b \leq a + 1$

Semaphore Properties — Liveness

- Semaphore S represented by s : *integer* := s_0 ($s_0 \geq 0$)
- $\mathtt{P}(S)$: $\langle\, s > 0 \rightarrow s := s - 1 \,\rangle$
  $\mathtt{V}(S)$: $\langle\, s := s + 1 \,\rangle$
- By fair process execution:

  $$\mathit{at}\ \mathtt{V}(S) \leadsto \mathit{after}\ \mathtt{V}(S)$$

- *Weakly fair semaphore*

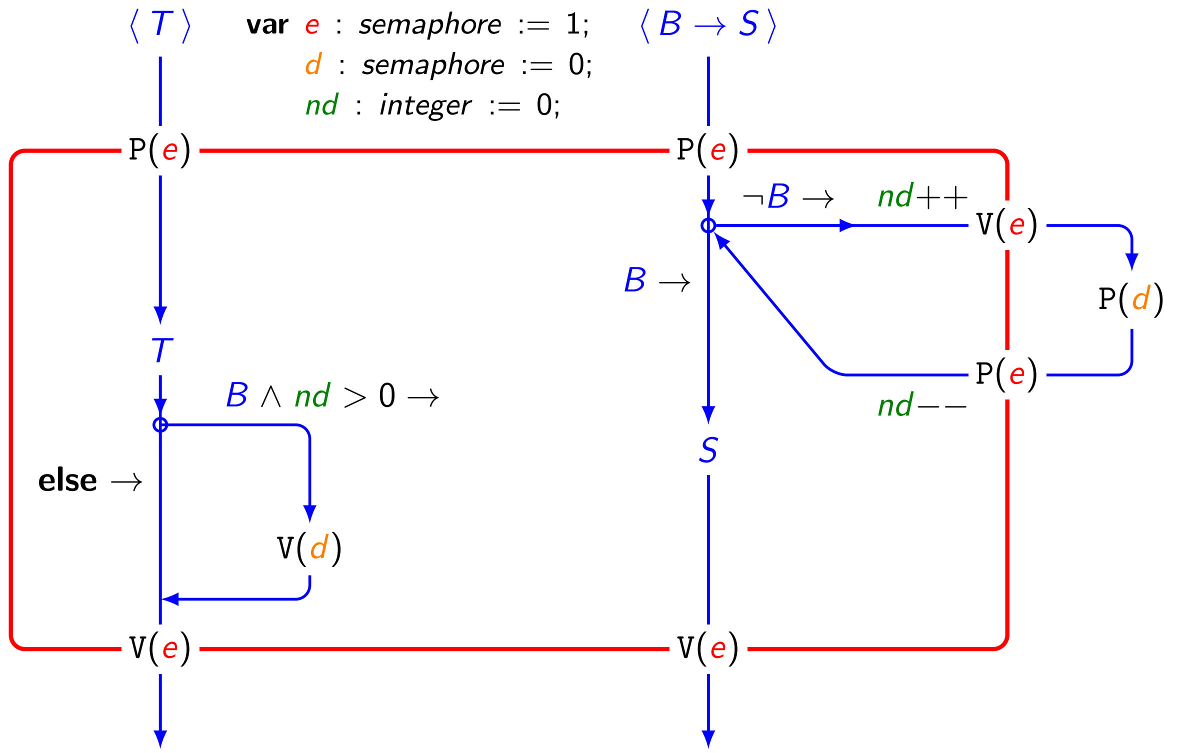
  $$(\mathit{at}\ \mathtt{P}(S) \wedge \Box s > 0) \leadsto \mathit{after}\ \mathtt{P}(S)$$

- Techniques: Busy-wait, semi busy-wait, wake-all
- *Strongly fair semaphore*

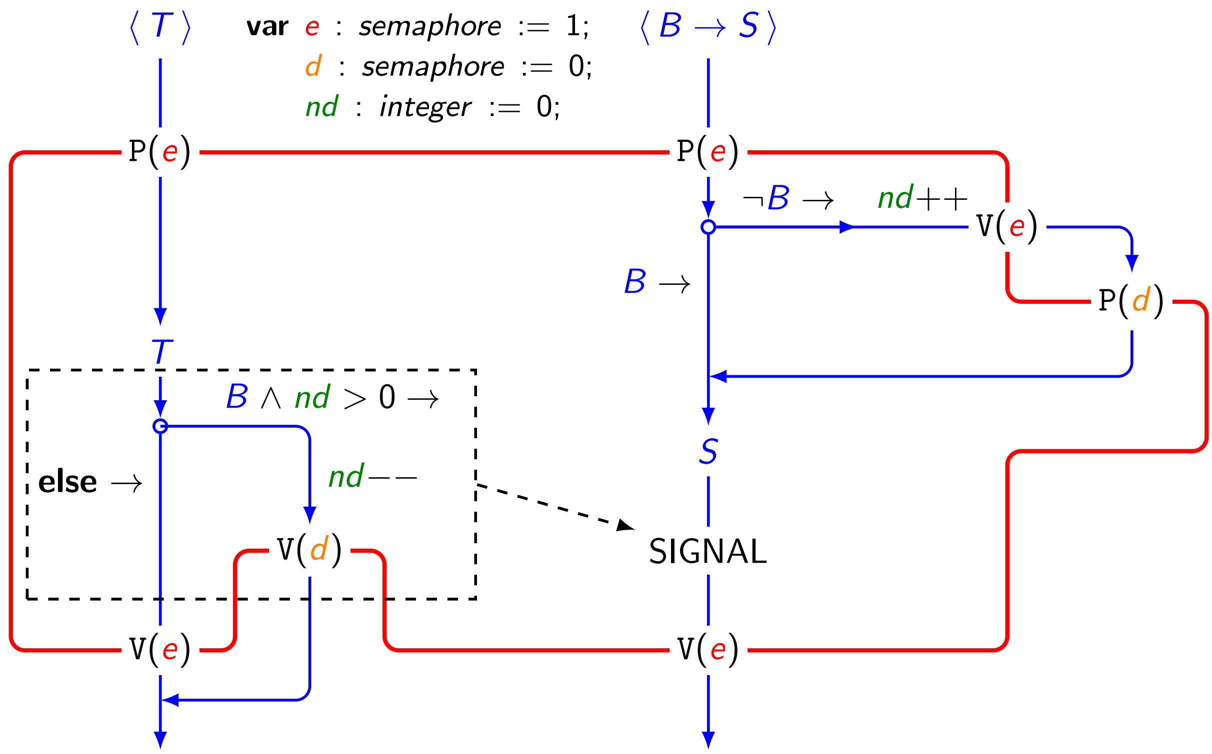
  $$(\mathit{at}\ \mathtt{P}(S) \wedge \Box\Diamond s > 0) \leadsto \mathit{after}\ \mathtt{P}(S)$$

- Techniques: FIFO, round-robin, aging

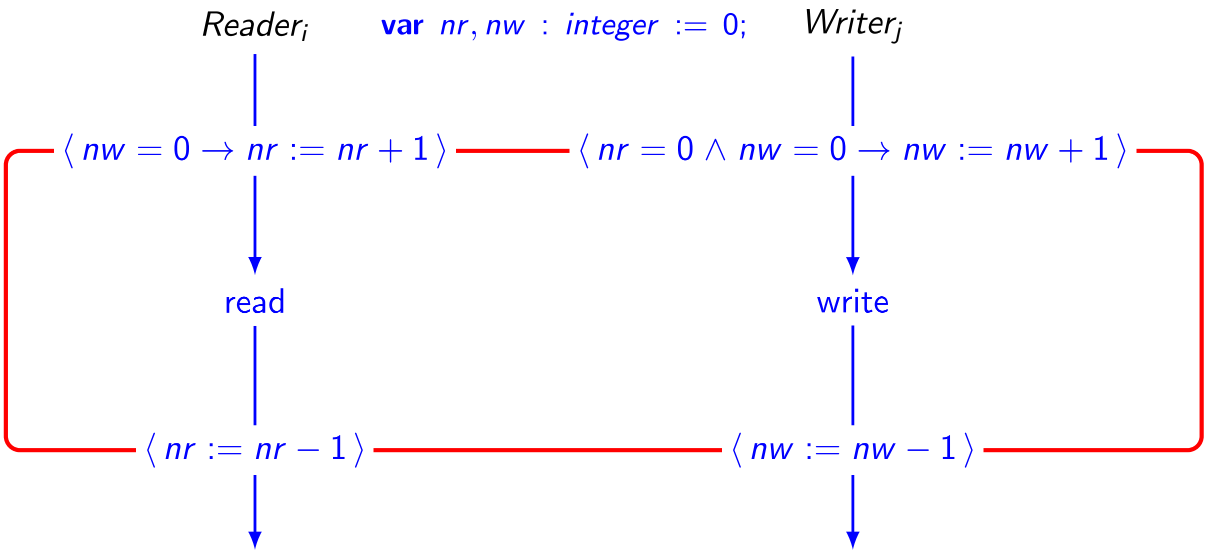
Passing-the-baton Technique (pre)



Passing-the-baton Technique



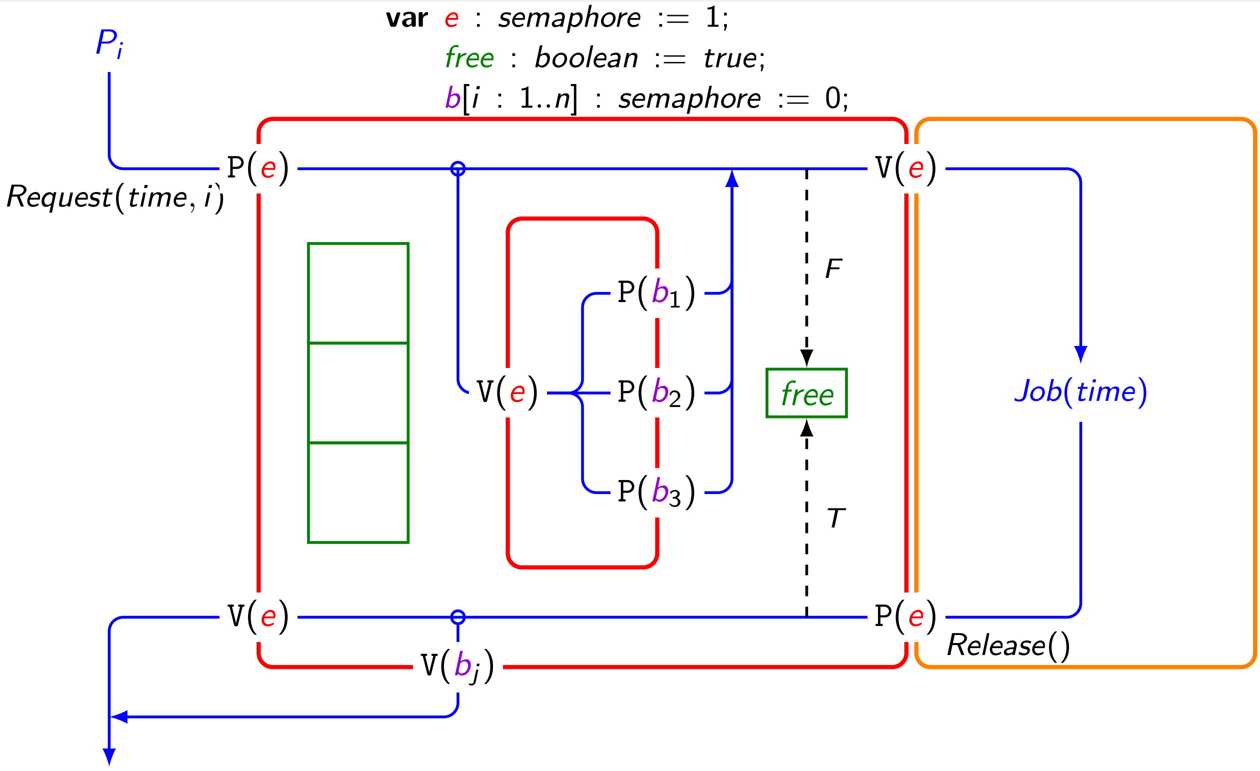
Reader/Writer with Coarse-grained Atomic Actions



- R/W invariant: $nw \leq 1 \wedge (nr = 0 \vee nw = 0)$
- Can then be implemented using *passing-the-baton* [Andrews 4.4.3]

Shortest-job-next

var e : *semaphore* := 1;
$free$: *boolean* := *true*;
$b[i : 1..n]$: *semaphore* := 0;



Shortest-job-next

- ```
  bool free = true;
  sem e = 1, b[n] = ([n] 0);  # for entry and delay
  typedef Pairs = set of (int, int);
  Pairs pairs = ∅;
  ## SJN:  pairs is an ordered set ∧ free ⇒ (pairs == ∅)
  ```
- ```
  request(time,id):
    P(e);
    if (!free) {
      insert (time,id) in pairs;
      V(e);          # release entry lock
      P(b[id]);      # wait to be awakened
    }
    free = false;
    V(e);    # optimized since free is false here
  ```
- ```
  release():
    P(e);
    free = true;
    if (P != ∅) {
      remove first pair (time,id) from pairs;
      V(b[id]);   # pass baton to process id
    }
    else V(e);
  ```
  ```

Java Semaphores

- Generalized Dijkstra semaphores
- Lots of *peek-and-poke* operations

Operations

- A Java `Semaphore` consists of:
  - A *permission count* s $\quad (s \geq 0)$
  - A queue of waiting threads
  - Queue may be *(strongly) fair* (FIFO)
- Operations

`s.acquire()`	$\langle\, s > 0 \rightarrow s := s - 1 \,\rangle$
`s.release()`	$\langle\, s := s + 1 \,\rangle$
`s.acquire(`n`)`	$\langle\, s \geq n \rightarrow s := s - n \,\rangle$
`s.release(`n`)`	$\langle\, s := s + n \,\rangle$

Pthreads Semaphores I

```
# include <pthread.h>      /* standard lines */
# include <semaphore.h>
# define SHARED 1
# include <stdio.h>

void *Producer(void *);  /* the two threads */
void *Consumer(void *);

sem_t empty, full;       /* global semaphores */
int data;                /* shared buffer     */
int numIters;

/* main() -- read command line and create threads */
int main(int argc, char *argv[]) {
  pthread_t pid, cid;      /* thread and attributes */
  pthread_attr_t attr;     /* descriptors           */
  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

  sem_init(&empty, SHARED, 1);  /* sem empty = 1 */
  sem_init(&full, SHARED, 0);   /* sem full = 0  */

  numIters = atoi(argv[1]);
  pthread_create(&pid, &attr, Producer, NULL);
  pthread_create(&cid, &attr, Consumer, NULL);
  pthread_join(pid, NULL);
  pthread_join(cid, NULL);
}
```

Pthreads Semaphores II

```
/* deposit 1, ..., numIters into the data buffer */
void *Producer(void *arg) {
  int produced;
  for (produced = 1; produced <= numIters; produced++) {
    sem_wait(&empty);
    data = produced;
    sem_post(&full);
  }
}

/* fetch numIters items from the buffer and sum them */
void *Consumer(void *arg) {
  int total = 0, consumed;
  for (consumed = 1; consumed <= numIters; consumed++) {
    sem_wait(&full);
    total = total + data;
    sem_post(&empty);
  }
  printf("the total is %d\n", total);
}
```