Semaphores

A shared integer variable, s, initialized to init, and manipulated only by two operations:

pass (*proberen*): $\mathtt{P}(\mathtt{s}) \stackrel{\text{df}}{=} \langle \mathtt{await}(\mathtt{s} > 0)\mathtt{s} = \mathtt{s} - 1 \rangle$

release (*verhogen*): $\mathtt{V}(\mathtt{s}) \stackrel{\text{df}}{=} \langle \mathtt{s} = \mathtt{s} + 1 \rangle$

- s is non-negative
- General semaphore: can take on any nonnegative value.
- Binary Semaphore: either 0 or 1.

Fairness: V may release a waiting process

- Weakly fair,
- Strongly fair, or
- FIFO

Unless otherwise stated we'll assume only weak fairness.

Inference Rules

$$\frac{(\mathrm{P} \wedge g > 0) \Rightarrow \mathrm{Q}_{g \leftarrow (g-1)}}{\{\mathrm{P}\}\mathtt{P}(\mathrm{g})\{\mathrm{Q}\}}$$

$$\frac{\mathrm{P} \Rightarrow \mathrm{Q}_{g \leftarrow (g+1)}}{\{\mathrm{P}\}\mathtt{V}(\mathrm{g})\{\mathrm{Q}\}}$$

Mutual Exclusion

Coarse-grained

```
int s = 1;

process CS[i = 1 to n] {
  while (true) {
    < await(s>0) s--; >
    critical section;
    < s++; >
    noncritical section;
  }
}
```

Fine-grained

```
sem s = 1;

process CS[i = 1 to n] {
  while (true) {
    P(s);
    critical section;
    V(s);
    noncritical section;
  }
}
```

Barrier Synchronization

signaling semaphore

- Used to signal event (i.e., arrival at some part of the code).
- Usually initialized to 0.

Use two semaphores per process pair:

- one signals arrival at barrier,
- another to control departure

Two process case:

```
sem arrive1 = 0; # Shared
sem arrive2 = 0; # Shared

process Worker1 {
  while (true) {
    code to implement task 1;
    V(arrive1);
    P(arrive2);
  }
}
```

```
process Worker2 {
  while (true) {
    code to implement task 2;
    V(arrive2);
    P(arrive1);
  }
}
```

Can be extended to n-processes by appropriate choice of semaphores.

Barrier Synchronization: Coordinator

- Workers signal arrival with V(done)
- Wait on P(continue[i])
- Coordinator waits on $n \times$ P(done)
- Releases all with V(continue[i])

Worker

```
sem done = 0;
sem continue[1:n] = ([n] 0);

process Worker[i = 1 to n] {
  while (true) {
    code to implement task i;
    V(done);
    P(continue[i]);
  }
}
```

Coordinator

```
process Coordinator {
  while (true) {
    for [i = 1 to n] P(done);
    for [i = 1 to n] V(continue[i]);
  }
}
```


Split Binary Semaphore

- Use semaphores to signal data state rather than process state.
- *split binary semaphore* — two or more binary semaphores that have the property that at most one is 1 at any time.
- Initially only one is 1.
- Invariant: $0 \leq s_0 + s_1 + \ldots + s_n \leq 1$
- In **every** execution path, a P operation on one semaphore is followed (eventually) by a V on a (possibly different) semaphore.
- Code between P and V executed in mutual exclusion.

Producers & Consumers

```
int buf;
sem empty = 1, full = 0;

process Producer[i = 1 to n] {
  while (true) {
    P(empty);
    deposit to buf;
    V(full);
  }
}
```

```
process Consumer[i = 1 to m] {
  while (true) {
    P(full);
    fetch from buf;
    V(empty);
  }
}
```


Semaphores as Counters

System with N (identical) resources that are to be shared.

- Use semaphore to represent number available,
- P to obtain one,
- V to release one.

Consider producer-consumer with bounded buffer of size N and multiple producers and consumers.

Producer and Consumer

```
int buf[0:N];
int front = 0; # next cell to read
int rear = 0;  # next cell to write
sem empty = N; # Num. empty cells
sem full = 0;  # Num. full cells
sem mutexA = 1;
sem mutexF = 1;

void Add(int x) {
  P(empty);
  P(mutexA);
  buf[rear] = x;
  rear = (rear + 1) % N;
  V(mutexA);
  V(full);
}
```

```
int Fetch() {
  P(full);
  P(mutexF);
  int result = buf[front];
  front = (front + 1) % N;
  V(mutexF);
  V(empty);
  return result;
}
```


Overlapping Shared Resources

Dining Philosophers

```
process Philosophers[i = 0 to n] {
  while (true) {
    think;
    acquire forks;
    eat;
    release forks;
  }
}
```

Wait-for cycle – two or more process such that every one is waiting for a resource held by another. (e.g., $p[0:n]$ such that $p[i]$ is waiting for something held by $p[(i+1)\%n]$ for all i.)

- A necessary condition for deadlock.
- Eliminate by asymetry.

Aside: Necessary Conditions for Deadlock

- Serially reusable resources shared under mutual exclusion
- Incremental acquisition
- No pre-emption
- Wait-for cycle

Readers/Writers Problem

- Several processes share a database,
- `Readers` — several can access concurrently.
- `Writers` — must have exclusive access.

Two solution forms:

1) Mutual exclusion — use semaphore for lock and count the readers.
   - First reader in acquires lock, last reader out releases it.
   - Writer acquires lock and releases when it's done.

2) Conditional synchronization — Passing the Baton

Reader-Writer Coarse Grained Solution

```
int nw := 0; # number of writers
int nr := 0; # number of readers
## INV: nw == 0 \/ (nw == 1 /\ nr = 0)
```

```
process Reader[i = 1 to M] {
  while (true) {
    < await(nw == 0) nr++; >
    read database
    < nr--; >
  }
}
```

```
process Writer[i = 1 to N] {
  while (true) {
    < await(nr == 0 && nw == 0) nw++; >
    write database
    < nw--; >
  }
}
```


Passing the Baton

A technique to implement general `await` statements using (split binary) semaphores:

- `sem e = 1;` — Control entry to atomic statements.
- For each condition (guard), `B`:
  - A semaphore — to delay processes that do `await(B)`
  - A counter — counts the number of delayed processes.

Global data

```
int nw := 0, nr := 0; # number of writers/readers
## INV: nw == 0 \/ (nw == 1 /\ nr = 0)

sem e := 1; # exclusive access
sem r := 0; # used to delay readers
sem w := 0; # used to delay writers
int dr := 0, dw := 0; # count of delayed readers/writers
```

Signal

```
if (nw == 0 and dr > 0) {
  dr = dr-1; V(r); # Awaken a reader
} else if (nr == 0 and nw == 0 and dw > 0) {
  dw = dw-1; V(w); # Awaken a writer
} else {
  V(e); # Release entry lock
}
```



```
process Reader[i = 1 to M] {
  while (true) {
    P(e);
    if (nw > 0) {
      dr++; V(e); P(r);
    }
    nr = nr + 1;
    SIGNAL;
    read database
    P(e);
    nr = nr - 1;
    SIGNAL;
  }
}
```

```
process Writer[j = 1 to N] {
  while (true) {
    P(e);
    if (nr > 0 or nw > 0) {
      dw++; V(e); P(w);
    }
    nw = nw + 1;
    SIGNAL;
    write database
    P(e);
    nw = nw - 1;
    SIGNAL;
  }
}
```