Critical Section

process CS[i = 1 to n] {
  while (true) {
    entry protocol;
    critical section;
    exit protocol;
    noncritical section;
  }
}

Assume process entering CS will eventually leave it.

Mutual Exclusion: At most one process in a critical section at a time.
Absence of Deadlock: If two or more processes are trying to enter, one will succeed.
Absence of Unnecessary Delay: Process gets to enter CS without unnecessary delay.
Eventual Entry: A process trying to enter CS will eventually succeed.

Note: \langle S \rangle is implemented by:

CSenter;
S
CSexit;

(Assuming that all other non-independent statements are similarly protected.)

Coarse Grained Solution

bool lock = false;
# bool in[1:n] = { false } -- 'thought' variable
## INV: \mid \{ j | in[j] \} \mid \leq 1 \land lock = (E)j.in[j]
process CS[i = 1 to n] {
  while (true) {
    \texttt{<await (!lock) lock = true;}
    # in[i] = true
  } \texttt{>}
  ## in[i]
  critical section;
  \texttt{< # in[i] = false}
  lock = false; \texttt{>}
  noncritical section;
  \texttt{>}
}

Hardware Solution: Test & Set

Modern CPUs offer instructions to aid mutual exclusion. Test-and-set is one:

\text{TS } r_i r_j \overset{df}= \langle r_i := M[r_j];~M[r_j] := \text{true;} \rangle

Implement above coarse grained solution as:

bool lock = false;
process CS[i = 1 to n] {
  while (true) {
    do { r1 := &lock;
      TS r0 r1;
    } while (r0);
    critical section;
    lock = false;
    noncritical section;
  } }
After-you Algorithm

```c
int t := 0;
## INV: t == 0 \lor t == 1
process P0 {
  while (true) {
    t := 1;
    < await (t == 0) >
    ## t == 0;
    critical section;
  }
}
```

- Enforces mutual exclusion
- Deadlock free
- Causes unnecessary delay
- Doesn’t ensure eventual entry

To prove mutual exclusion we want to find assertions $A_0$ and $A_1$ such that:

- $A_0$ is true whenever $P_0$ is in its critical section.
- $A_1$ is true whenever $P_1$ is in its critical section.
- Both can’t be true at once: $\neg (A_0 \land A_1)$

**A first try:**

$A_0 \overset{df}{=} r[0] \land \neg r[1]$

$A_1 \overset{df}{=} r[1] \land \neg r[0]$

Is there interference?

**A second try:**

Introduce a "thought variable" (auxiliary variable) $t$ such that if $r[0] \land r[1]$ then $t = i$ indicates that process $i$ was the first to set its request flag.

```c
bool r[2] := {false, false};
int t := 0; # thought variable
process P0 {
  while (true) {
    r[0] := true; t := 1;
    < await (!r[1]) >
    ## A0
    critical section;
    r[0] := false;
  }
}
```

- Enforces mutual exclusion
- Deadlocks

Safe-Sluice Algorithm

```c
bool r[2] := {false, false};
process P0 {
  r[0] := true;
  < await (!r[1]) >
  ## A0
  critical section;
  r[0] := false;
}
```

```c
process P1 {
  r[1] := true;
  < await (!r[0]) >
  ## A1
  critical section;
  r[1] := false;
}
```

- Enforces mutual exclusion
- Deadlocks

**A second try:**

```c
bool r[2] := {false, false};
int t := 0; # thought variable
process P0 {
  while (true) {
    < r[0] := true; t := 1; >
    < await (!r[1]) >
    ## A0
    critical section;
    r[0] := false;
  }
}
```

```c
process P1 {
  < r[1] := true; t := 0 >
  < await (!r[0]) >
  ## A1
  critical section;
  r[1] := false;
}
```

- Enforces mutual exclusion
- Deadlocks

**A first try:**

$A_0 \overset{df}{=} r[0] \land \neg r[1]$

$A_1 \overset{df}{=} r[1] \land \neg r[0]$

Is there interference?

```c
A_0 \overset{df}{=} r[0] \land \neg r[1]
A_1 \overset{df}{=} r[1] \land \neg r[0]
```

```c
A_0 \overset{df}{=} r[0] \land (\neg r[1] \lor t = 0)
A_1 \overset{df}{=} r[1] \land (\neg r[0] \lor t = 1)
```

- interference?
- $A_0 \land A_1 \Leftrightarrow false$?
### Eliminating Deadlock

We can use $t$ to eliminate the deadlock in Safe-sluice:

```plaintext
bool r[2] := {false, false};
int t := 0;  # no longer just a thought variable

process P0 {
    < r[0] := true; t := 1; >  # A0
    < await (!r[1] || t == 0) >
    critical section;
    r[0] := false;
}

process P1 {
    < r[1] := true; t := 0 >  # A1
    < await (!r[0] || t == 1) >
    critical section;
    r[1] := false;
}
```

### Splitting the atomic assignment — Peterson's algorithm

If we do it right we don’t need to combine the two assignment statements into an atomic action:

```plaintext
bool r[2] := {false, false};
int t := 0;  # turn indicator

process P0 {
    r[0] := true;
    t := 1;
    < await (!r[1] || t == 0) >
    ## B0
    critical section;
    r[0] := false;
}

process P1 {
    r[1] := true;
    t := 0;
    < await (!r[0] || t == 1) >
    ## B1
    critical section;
    r[1] := false;
}
```

We need new assertions $B0$ and $B1$ (why?)

### Spin Loops

Note that the `await` condition `!r[1] || t == 0` does not satisfy the AMO property. Despite this we can still implement it using a spin loop. Think of it this way:

```plaintext
loop {
    exit when !r[1]
    exit when t == 0
}
```

This is implemented as

```plaintext
while (r[1] && t != 0) /* spin */ ;
```

Can we implement `< await(a && b) >` as `while (!a || !b) /* spin */ ;`?
Full Proof Outline

To be complete we should put in all the assertions and show non-interference.

bool r[2] := {false, false};
int t := 0; # turn indicator
bool n[2] = {false, false}

process P0 {
## true
< r[0] := true; n[0] := true>
## r[0] && n[0]
< t := 1; n[0] := false >
## r[0] && !n[0]
< await (!r[1] || t == 0) >
## B0
critical section;
r[0] := false;
}

process P1 {
## true
## r[1] && n[1]
< t := 0; n[1] := false; >
## r[1] && !n[1]
< await (!r[0] || t == 1) >
## B1
critical section;
r[1] := false;
}

Bakery Algorithm

Invariant:

∀i, 1 ≤ i ≤ n ⇒
( (CS[i] in its critical section ) ⇒ (turn[i] > 0 ∧
∀j, (1 ≤ j ≤ n ∧ j ≠ i) ⇒ (turn[j] = 0 ∨ turn[i] < turn[j]))

int turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
    while (true) {
        < turn[i] = max(turn[1:n]) + 1; >
        for [j = 1 to n st j != i]
            < await (turn[j] == 0 or turn[i] < turn[j]) ; >
        critical section;
        turn[i] = 0;
        noncritical section;
    }
}

Barrier Synchronization

Typical structure for parallel iterative algorithms:

for (int r = 1; r <= 20; r++) {
    turn.val[pid] = 1;
    turn.val[pid] = turn.max() + 1;
    for (int j = 0; j < n; j++) {
        if (j != pid) {
            while (turn.val[j] != 0 &&
                (turn.val[pid] > turn.val[j] ||
                (turn.val[pid] == turn.val[j] && pid > j)))
                Thread.sleep((int)Math.round(Math.random()*1));
        }
    }
    // Critical section
    // Exit protocol
    turn.val[pid] = 0;
    // noncritical section
}
### Mutual Inclusion

```c
int s[n] := {-1}; int c[n] := {-1};

process Pi {
    for [r := 0 to n] {
        s[i] := s[i] + 1;
        Round(i, r);
        c[i] := c[i] + 1;
        Barrier
    }
}

process Pj {
    for [r := 0 to n] {
        s[j] := s[j] + 1;
        Round(j, r);
        c[j] := c[j] + 1;
        Barrier
    }
}
```

Working: \( s[i] > c[i] \)

In barrier: \( s[i] == c[i] \)

- While process \( i \) is working on round \( k \), process \( j \) must be finished round \( k - 1 \) and not yet started round \( k + 1 \).

- Desired invariant: \( s[i] > c[i] \Rightarrow \forall j, c[j] \geq c[i] \land s[j] \leq s[i] \)

### Shared Counter

```c
process Worker[i = 1 to n] {
    while (true) {
        code to implement task i;
        < count = count + 1; >
        < await (count == n); >
    }
}
```

How to ensure that \( count = 0 \) at the start of each iteration?

### Flags and Coordinators

```c
int arrive[1:n] = {0};
int continue[1:n] = {0};

process Worker[i = 1 to n] {
    while (true) {
        code to implement task i;
        arrive[i] = 1;
        < await (continue[i] == 1); >
        continue[i] = 0;
    }
}

process Coordinator {
    while (true) {
        for [i = 1 to n] {
            < await (arrive[i] == 1); >
            arrive[i] = 0;
        }
        for [i = 1 to n] continue[i] = 1;
    }
}
```

### Flag Synchronization Principles

- A process that waits for a synchronization flag to be set should be the one to clear the flag.
- A flag should not be set until it is known to be clear.

#### Inefficiencies
- Extra process for Coordinator
- Coordinator is slower for more processes.

#### Solutions
- Combining Tree Barrier
- Symmetric Barrier
### Data Parallel Algorithms

**Parallel Prefix:** \( \forall i, 0 \leq i < n \Rightarrow \text{sum}[i] = \sum_{j=0}^{i} a[j] \)

```c
int a[n], sum[n], old[n]

process Sum[i = 0 to n-1] {
    int d = 1; # distance
    sum[i] = a[i]; # initialize to a
    barrier(i);
    while (d < n) {
        old[i] = sum[i];
        barrier(i);
        if ((i-d) >= 0) sum[i] += old[i-d];
        barrier(i);
        d += d; # double distance
    }
}
```

### Jacobi Iteration (Laplace’s eqn):

```c
int grid[n+1,n+1], newgrid[n+1,n+1];
bool converged = false;

process Grid[i = 1 to n, j = 1 to n] {
    while (!converged) {
        newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] +
                        grid[i,j-1] + grid[i,j+1]) / 4;
        converged = (test for convergence)
        barrier(i);
        grid[i,j] = newgrid[i,j];
        barrier(i);
    }
}
```

### Bag of Tasks

```c
while (bag is not empty) {
    get task from the bag;
    execute the task, possibly generating new ones;
}
```

- Task is independent unit of work.
- Bag represents collection of tasks.
- Scalable — set number of workers to number of processors.
- Load balanced — if a task takes longer, other workers will do more tasks.

### Example: Adaptive Quadrature

```c
process Worker[w = 1 to PR] {
    double left, right, fleft, fright, lrarea;
    double mid, fmid, larea, rarea;
    bool done = false;
    while (!done) {
        < idle++;
        done = (idle == PR && bag is empty); >
        if (!done) {
            < await (size > 0)
            get task (left, right, fleft, fright, lrarea)
            from bag;
            idle--; >
            mid = (left+right)/2;
            fmid = f(mid);
            larea = (fleft + fmid) * (mid - left) / 2;
            rarea = (fmid + fright) * (right - mid) / 2;
            if ((abs(larea+rarea) - lrarea) > EPSILON) {
```
put (left, mid, fleft, fmid, larea) into bag;
    put (mid, right, fmid, fright, rarea) into bag;
} else {
    total += lrarea;
}