Critical Section

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

Assume process entering CS will evenutally leave it.

Mutual Exclusion: At most one process in a critical section at at 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: $\langle \mathtt{S} \rangle$ 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: |{ j | in[j] }| <= 1 /\ lock = (E)j.in[j]
process CS[i = 1 to n] {
  while (true) {
    <await (!lock) lock = true;
    # in[i] = true
    >
    ## in[i]
    critical section;
    < # in[i] = false
    lock = false; >
    noncritical section;
  }
}
```


Hardware Solution: Test & Set

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

$$\mathtt{TS}\ \mathtt{r_i}\ \mathtt{r_j} \stackrel{\text{df}}{=} \langle \mathtt{r_i} := \mathtt{M}[\mathtt{r_j}];\ \mathtt{M}[\mathtt{r_j}] := \mathtt{true}; \rangle$$

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

```
int t := 0;
## INV: t == 0 \/ t == 1

process P0 {                    process P1 {
    while (true) {                  while (true)
      t := 1;                         t := 0;
      < await (t == 0) >              < await (t == 1) >
      ## t == 0;                      ## t == 1;
      critical section;               critical section;
    }                               }
}                               }
```

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

Safe-Sluice Algorithm

```
bool r[2] := {false, false};

process P0 {                    process P1 {
    r[0] := true;                   r[1] := true;
    < await (!r[1]) >               < await (!r[0]) >
    ## A0                           ## A1
    critical section;               critical section;
    r[0] := false;                  r[1] := false;
}                               }
```

- Enforces mutual exclusion
- Deadlocks

To prove mutual exclusion we want to find assertions A0 and A1 such that:

- A0 is true whenever P0 is in its critical section.
- A1 is true whenever P1 is in its critical section.
- Both can't be true at once: $\neg(\mathtt{A0} \wedge \mathtt{A1})$

A first try:

$\mathtt{A0} \stackrel{\mathrm{df}}{=} \mathtt{r}[0] \wedge \neg\mathtt{r}[1]$ $\qquad$ $\mathtt{A1} \stackrel{\mathrm{df}}{=} \mathtt{r}[1] \wedge \neg\mathtt{r}[0]$

Is there interference?

A second try:

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

```
bool r[2] := {false, false};
int t := 0; # thought variable

process P0 {                    process P1 {
    < r[0] := true; t := 1; >       < r[1] := true; t := 0 >
    < await (!r[1]) >               < await (!r[0]) >
    ## A0                           ## A1
    critical section;               critical section;
    r[0] := false;                  r[1] := false;
}                               }
```

$\mathtt{A0} \stackrel{\mathrm{df}}{=} \mathtt{r}[0] \wedge (\neg\mathtt{r}[1] \vee \mathtt{t} = 0)$ $\qquad$ $\mathtt{A1} \stackrel{\mathrm{df}}{=} \mathtt{r}[1] \wedge (\neg\mathtt{r}[0] \vee \mathtt{t} = 1)$

- interference?
- $\mathtt{A0} \wedge \mathtt{A1} \Leftrightarrow \mathit{false}$?

Eliminating Deadlock

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

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

process P0 {                     process P1 {
    < r[0] := true; t := 1; >        < r[1] := true; t := 0 >
    < await (!r[1] || t == 0) >      < await (!r[0] || t == 1) >
    ## A0                            ## A1
    critical section;                critical section;
    r[0] := false;                   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:

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

process P0 {                     process P1 {
    r[0] := true;                    r[1] := true;
    t := 1;                          t := 0
    < await (!r[1] || t == 0) >      < await (!r[0] || t == 1) >
    ## B0                            ## B1
    critical section;                critical section;
    r[0] := false;                   r[1] := false;
}                                }
```

We need new assertions B0 and B1 (why?)

Let's introduce another thought variable $\mathtt{n}[i]$ to indicate when $\mathtt{P}i$ is between the two assingments:

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

process P0 {                     process P1 {
    < r[0] := true; n[0] := true>    < r[1] := true; n[1] := true; >
    < t := 1; n[0] := false >        < t := 0; n[1] := false; >
    < await (!r[1] || t == 0) >      < await (!r[0] || t == 1) >
    ## B0                            ## B1
    critical section;                critical section;
    r[0] := false;                   r[1] := false;
}                                }
```

$\mathtt{B0} \stackrel{\text{df}}{=} \mathtt{r}[0] \wedge \neg\mathtt{n}[0] \wedge (\neg\mathtt{r}[1] \vee \mathtt{t} = 0 \vee \mathtt{n}[1])$
$\mathtt{B1} \stackrel{\text{df}}{=} \mathtt{r}[1] \wedge \neg\mathtt{n}[1] \wedge (\neg\mathtt{r}[0] \vee \mathtt{t} = 1 \vee \mathtt{n}[0])$

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:

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

Clearly when the loop exits the assertion is true.

This is implemented as
`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] := true; n[1] := 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:

$$\forall i, 1 \le i \le n \Rightarrow \Big((\texttt{CS}[i] \text{ in its critical section }) \Rightarrow (\texttt{turn}[i] > 0 \wedge \forall j, (1 \le j \le n \wedge j \ne i) \Rightarrow (\texttt{turn}[j] = 0 \vee \texttt{turn}[i] < \texttt{turn}[j])) \Big)$$

```
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;
  }
}
```



```
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
}
```


Barrier Synchronization

Typical structure for parallel itterative algorithms:

```
process Worker[i = 1 to n] {
  while (true) {
    code to implement task i;
    wait for all n tasks to complete;
  }
}
```

Mutual Inclusion

```
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: $\mathtt{s}[i] > \mathtt{c}[i]$
In barrier: $\mathtt{s}[i] == \mathtt{c}[i]$
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: $\mathtt{s}[i] > \mathtt{c}[i] \Rightarrow \forall j, \mathtt{c}[j] \geq \mathtt{c}[i] \wedge \mathtt{s}[j] \leq \mathtt{s}[i]$

Shared Counter

```
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 itteration?

Flags and Coordinators

```
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 Priciples

- 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: $\forall i, 0 \leq i < n \Rightarrow \mathtt{sum}[i] = \sum_{j=0}^{i} \mathtt{a}[j]$

```
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):

```
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

```
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 tasks takes longer, other workers will do more tasks.

Example: Adaptive Quadrature

```
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;
    }
  }
} }
```