Locks and Barriers

Andrews Chapter 03

Critical Section Problem

- Mutual Exclusion.
- Absence of Deadlock (Livelock).
- Absence of Unnecessary Delay.
- Eventual Entry.

Three safety properties, one liveness property.

```
bool in1 = false, in2 = false;
## MUTEX:  ¬(in1 ∧ in2) -- global invariant

process CS1 {
  while (true) {
    ⟨await (!in2) in1 = true;⟩  /* entry */
    critical section;
    in1 = false;                /* exit  */
    noncritical section;
  }
}

process CS2 {
  while (true) {
    ⟨await (!in1) in2 = true;⟩  /* entry */
    critical section;
    in2 = false;                /* exit  */
    noncritical section;
  }
}
```

Figure 3.1 Critical section problem: Coarse-grained solution.

```
bool lock = false;

process CS1 {
  while (true) {
    ⟨await (!lock) lock = true;⟩  /* entry */
    critical section;
    lock = false;                  /* exit */
    noncritical section;
  }
}

process CS2 {
  while (true) {
    ⟨await (!lock) lock = true;⟩  /* entry */
    critical section;
    lock = false;                  /* exit */
    noncritical section;
  }
}
```

Figure 3.2 Critical sections using locks.

Test and Set

```
bool TS(bool lock) {
  ⟨
  bool initial = lock;
  lock = true;
  return initial;
  ⟩
}
```

```
bool lock = false;                  /* shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (TS(lock)) skip;          /* entry protocol */
    critical section;
    lock = false;                   /* exit protocol  */
    noncritical section;
  }
}
```

Figure 3.3 Critical sections using Test and Set.

- In spin-lock solution, exit protocol simply resets shared variables.

```
bool lock = false;                /* shared lock */
process CS[i = 1 to n] {
  while (true) {
    while (lock) skip;            /* entry protocol */
    while (TS(lock)) {
      while (lock) skip;
    }
    critical section;
    lock = false;                 /* exit protocol  */
    noncritical section;
  }
}
```

Figure 3.4 Critical sections using Test and Test and Set.

Implementing Await Statements

- Any critical section solution can be used to implement unconditional atomic actions:

```
⟨ S; ⟩
```

```
CSenter;
S;
CSexit;
```

- Provided all other code that could interfere with variables in `S` are also protected similarly.
- This was what we did using semaphores as **mutex**es.

Implementing Await Statements

- How should we add `await`?

```
⟨ await(B) s; ⟩
```

```
CSenter;
while (B) { ??? }
S;
CSexit;
```

- If we don't do anything, deadlock is guaranteed since all processes are blocked.

Implementing Await Statements

- $\langle$ `await(B) s;` $\rangle$

```
CSenter;
while (B) {
  CSexit;
  CSenter;
}
S;
CSexit;
```

- Correct but inefficient.
- Good chance the scheduler will not be very fair.

Implementing Await Statements

- $\langle$ `await(B) s;` $\rangle$

```
CSenter;
while (B) {
  CSexit;
  Delay;
  CSenter;
}
S;
CSexit;
```

- Gives more chance for other processes to change `B`
- Used in Ethernet **binary exponential backoff protocol.**
- Shown to be useful in critical section entry protocols, too.

Critical Sections: Fair Solutions

- Spin-lock solutions we've seen require a strongly fair scheduler.
- This is impractical.
- Three user-defined critical section protocols, only requiring weak fairness:
  - Tie breaker algorithm
  - Ticket algorithm
  - Bakery algorithm

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
  while (true) {
    last = 1; in1 = true;    /* entry protocol */
    〈await (!in2 or last == 2);〉
    critical section;
    in1 = false;             /* exit protocol */
    noncritical section;
  }
}
process CS2 {
  while (true) {
    last = 2; in2 = true;    /* entry protocol */
    〈await (!in1 or last == 1);〉
    critical section;
    in2 = false;             /* exit protocol */
    noncritical section;
  }
}
```

Figure 3.5 Two-process tie-breaker algorithm: Coarse-grained solution.

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
  while (true) {
    last = 1; in1 = true;    /* entry protocol */
    while (in2 and last == 1) skip;
    critical section;
    in1 = false;             /* exit protocol */
    noncritical section;
  }
}
process CS2 {
  while (true) {
    last = 2; in2 = true;    /* entry protocol */
    while (in1 and last == 2) skip;
    critical section;
    in2 = false;             /* exit protocol */
    noncritical section;
  }
}
```

Figure 3.6 Two-process tie-breaker algorithm: Fine-grained solution.

```
int in[1:n] = ([n] 0), last[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    for [j = 1 to n] {        /* entry protocol */
      /* remember process i is in stage j and is last */
      last[j] = i; in[i] = j;
      for [k = 1 to n st i != k] {
        /* wait if process k is in higher numbered stage
           and process i was the last to enter stage j */
        while (in[k] >= in[i] and last[j] == i) skip;
      }
    }
    critical section;
    in[i] = 0;                /* exit protocol */
    noncritical section;
  }
}
```

Figure 3.7 The **n**-process tie-breaker algorithm.

TICKET

$$
\begin{array}{l}
\text{next} > 0 \\
\qquad\qquad \wedge \\
(\forall_{1 \le i \le n} : \\
\qquad\qquad (\text{CS[i] in its critical section}) \Rightarrow (\text{turn[i]} == \text{next}) \\
\qquad\qquad \wedge \\
\qquad\qquad (\text{turn[i]} > 0) \Rightarrow (\forall_{1 \le j \le n, j \ne i} \text{turn[i]} \ne \text{turn[j]}) \\
\qquad\qquad)
\end{array}
$$

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
## predicate TICKET is a global invariant (see text)
process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = number; number = number + 1;⟩
    ⟨await (turn[i] == next);⟩
    critical section;
    ⟨next = next + 1;⟩
    noncritical section;
  }
}
```

Figure 3.8 The ticket algorithm: Coarse-grained solution.

Fetch and Add

```
FA(var, incr):
  ⟨ int tmp = var; var = var + incr; return(tmp); ⟩
```

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
process CS[i = 1 to n] {
  while (true) {
    turn[i] = FA(number,1);      /* entry protocol */
    while (turn[i] != next) skip;
    critical section;
    next = next + 1;             /* exit protocol  */
    noncritical section;
  }
}
```

Figure 3.9 The ticket algorithm: Fine-grained solution.

BAKERY

$$
\begin{aligned}
&(\forall_{1 \le i \le n} \\
&\qquad (\mathrm{CS[i]\ in\ critical\ section}) \Rightarrow (\mathrm{turn[i]} > 0) \\
&\qquad\qquad \wedge \\
&\qquad\qquad (\forall_{1 \le j \le n, j \ne i} \mathrm{turn[j]} = 0 \vee \mathrm{turn[i]} < \mathrm{turn[j]}) \\
&\quad)
\end{aligned}
$$

- Note: errata in book.

```
int turn[1:n] = ([n] 0);
## predicate BAKERY is a global invariant -- see text
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;
  }
}
```

Figure 3.10 The bakery algorithm: Coarse-grained solution.

- Entry protocol is difficult to implement.
- To understand the solution, start with a two-process solution.

Entry protocols, first try

CS1

```
turn1 = turn2 + 1;
while (turn2 != 0 and turn1 > turn2) skip;
```

CS2

```
turn2 = turn1 + 1;
while (turn1 != 0 and turn2 > turn1) skip;
```

- Both could set their turns to 1 at the same time.
- Both could enter their critical sections at the same time.

Entry protocols, asymmetry is slight improvement

CS1

```
turn1 = turn2 + 1;
while (turn2 != 0 and turn1 > turn2) skip;
```

CS2

```
turn2 = turn1 + 1;
while (turn1 != 0 and turn2 >= turn1) skip;
```

- Still possible for both to enter critical sections.
- `CS2` can "race by" `CS1`
- Called a **race condition**

Entry protocols

CS1

```
turn1 = 1; turn1 = turn2 + 1;
while (turn2 != 0 and turn1 > turn2) skip;
```

CS2

```
turn2 = 1; turn2 = turn1 + 1;
while (turn1 != 0 and turn2 >= turn1) skip;
```

- Works, but not symmetric

Generalized "Less Than"

$$
\begin{aligned}
(a,b) > (c,d) \;&= \text{true} \;\; (a > c) \vee (a = c \wedge b > d) \\
&= \text{false} \;\; \text{otherwise}
\end{aligned}
$$

Symmetric Entry Protocols

CS1

```
turn1 = 1; turn1 = turn2 + 1;
while (turn2 != 0 and (turn1,1) > (turn2,2)) skip;
```

CS2

```
turn2 = 1; turn2 = turn1 + 1;
while (turn1 != 0 and (turn2,2) >= (turn1,1)) skip;
```

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

Figure 3.11 Bakery algorithm: Fine-grained solution.

Barrier Synchronization

- Inefficient solution, too many tasks starting and stopping:

```
while (true) {
  co [i = 1 to n]
    code for task i
  oc
}
```

- **Much more costly to create and destroy processes than to synchronize them.**
- More efficient model:

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

```
int count = 0;
process Worker[i = 1 to n] {
  while (true) {
    code to implement task i;
    ⟨count = count + 1;⟩
    ⟨await (count == n);⟩
  }
}
```

Simple counter barrier in display (3.11)

- Can implement barrier with:

```
FA(count, 1);
while (count != n) skip;
```

- Problem is resetting `count` and looping
- `count` is a global variable for each process

Flags and Coordinators

- Distribute `count` over `arrive[1:n]`
- Global invariant becomes:

```
count == (arrive[1] + ...  + arrive[n])
```

- Waiting on this is just as bad:

```
⟨await ((arrive[1] + ...  + arrive[n]) == n); ⟩
```

- Use a coordinator task.
- Task i

```
arrive[i] = 1;
⟨ await (continue[i] == 1);
```

- Coordinator

```
for [i = 1 to n] ⟨ await (arrive[i] == 1); ⟩
for [i = 1 to n] continue[i] = 1;
```

Flag Synchronization Principles

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

```
int arrive[1:n] = ([n] 0),  continue[1:n] = ([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;
  }
}
```

Figure 3.12 Barrier synchronization using a coordinator process.

- Avoids memory contention.
- Is not symmetric.
- Coordinator spends most of its time waiting.
- Tasks have a linear time wait for coordinator.

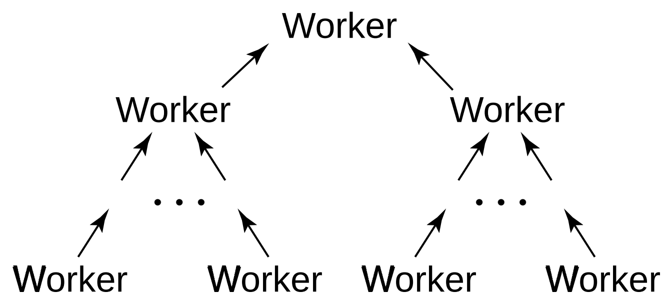

Figure 3.13 Tree-structured barrier.

- **Combining tree barrier.**

leaf node **L**:

```
arrive[L] = 1;
⟨await (continue[L] == 1);⟩
continue[L] = 0;
```

interior node **I**:

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
arrive[I] = 1;
⟨await (continue[I] == 1);⟩
continue[I] = 0;
continue[left] = 1; continue[right] = 1;
```

root node **R**:

```
⟨await (arrive[left] == 1);⟩
arrive[left] = 0;
⟨await (arrive[right] == 1);⟩
arrive[right] = 0;
continue[left] = 1; continue[right] = 1;
```

Figure 3.14 Barrier synchronization using a combining tree.

- More symmetric, each task does some real computation.
- But still three different kinds of nodes.

```
/* barrier code for worker process W[i] */
⟨await (arrive[i] == 0);⟩  /* key line -- see text */
arrive[i] = 1;
⟨await (arrive[j] == 1);⟩
arrive[j] = 0;

/* barrier code for worker process W[j] */
⟨await (arrive[j] == 0);⟩  /* key line -- see text */
arrive[j] = 1;
⟨await (arrive[i] == 1);⟩
arrive[i] = 0;
```

Two-process symmetric barrier in display (3.15)

- A symmetric two-process barrier.

  Wait clearing own flag.

  Set own flag.

  Wait setting other flag.

  Clear other flag.

- First line is necessary to prevent a process racing around.

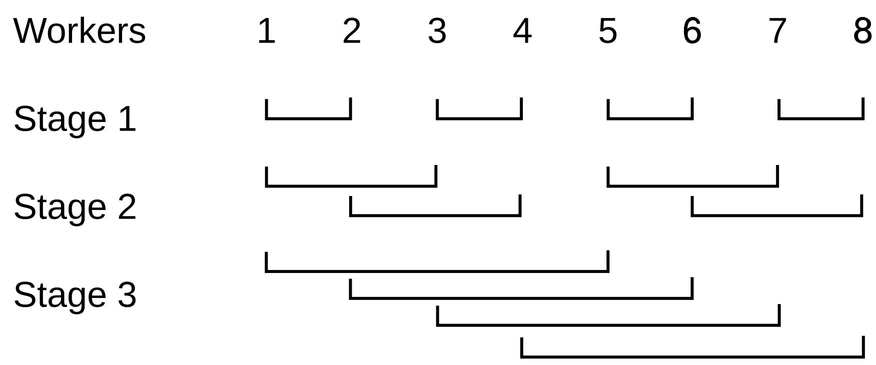

Figure 3.15 Butterfly barrier for 8 processes.

- Combining 2-process synchronization.
- At stage s synchronize with process 2^{s-1} away.
- n must be power of 2.

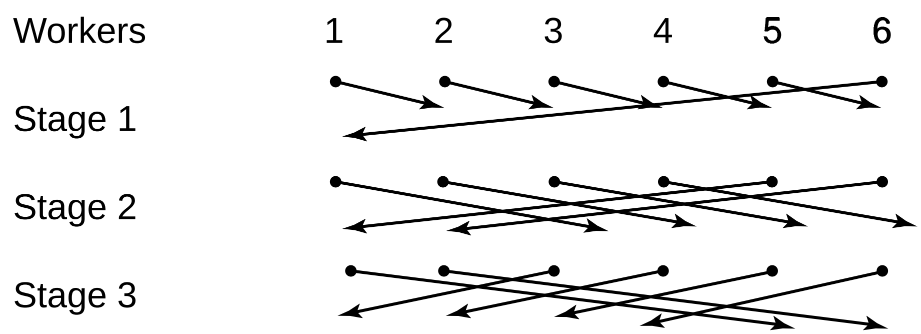

Figure 3.16 Dissemination barrier for 6 processes.

- At stage s synchronize with process 2^{s-1} away.
- **Dissemination barrier:**

  Set arrival flag of worker to right.

  Wait on own flag.

  Clear own flag.

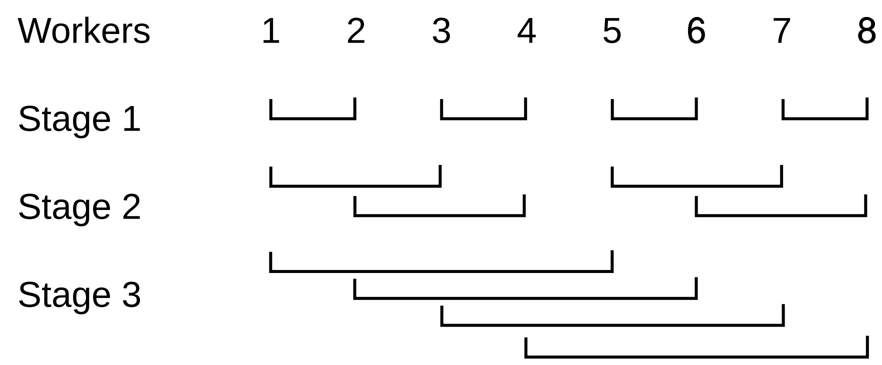

Figure 3.15 Butterfly barrier for 8 processes.

- Need to avoid global flags:

  Suppose 1 finishes, but 2 is slow.

  Now 3 and 4 finish.

  Now 3 tries to synchronize with 1, and thinks it is ready.

- Could use different flags for each level.
- Or use integer flags.

Data Parallel Algorithms

- Many processes execute the same code and work on different parts of shared data.
- Usually associated with parallel hardware, *e.g.* graphics cards.
- Barrier synchronization usually in hardware.
- Can be useful on asynchronous processors when granularity of the processes is large enough to compensate for synchronization overhead.

Partial sums of an array

Sequential solution

```
sum[0] = a[0];
for [i = 1 to n-1]
  sum[i] = sum[i-1] + a[i]
```

initial values of a	1	2	3	4	5	6
partial sums	1	3	6	10	15	21

```
int a[n], sum[n], old[n];
process Sum[i = 0 to n-1] {
  int d = 1;
  sum[i] = a[i];   /* initialize elements of sum */
  barrier(i);
  ## SUM:  sum[i] = (a[i-d+1] + ... + a[i])
  while (d < n) {
    old[i] = sum[i];   /* save old value */
    barrier(i);
    if ((i-d) >= 0)
      sum[i] = old[i-d] + sum[i];
    barrier(i);
    d = d+d;   /* double the distance */
  }
}
```

Figure 3.17 Computing all partial sums of an array.

initial values of a	1	2	3	4	5	6
sum after distance 1	1	3	5	7	9	11
sum after distance 2	1	3	6	10	14	18
sum after distance 4	1	3	6	10	15	21

- A $\log(n)$ concurrent solution using **doubling**.

```
int link[n], end[n];
process Find[i = 0 to n-1] {
  int new, d = 1;
  end[i] = link[i];   /* initialize elements of end */
  barrier(i);
  ## FIND:  end[i] == index of end of the list
  ##        at most 2^(d-1) links away from node i
  while (d < n) {
    new = null;      /* see if end[i] should be updated */
    if (end[i] != null and end[end[i]] != null)
      new = end[end[i]];
    barrier(i);
    if (new != null)    /* update end[i] */
      end[i] = new;
    barrier(i);
    d = d + d;          /* double the distance */
  }
}
```

Figure 3.18 Finding the end of a serially linked list.

- Find the end of linked list in $\log(n)$ time.

```
real grid[n+1,n+1], newgrid[n+1,n+1];
bool converged = false;
process Grid[i = 1 to n, j = 1 to n] {
  while (not converged) {
    newgrid[i,j] = (grid[i-1,j] + grid[i+1,j] +
                    grid[i,j-1] + grid[i,j+1]) / 4;
    check for convergence as described in the text;
    barrier(i);
    grid[i,j] = newgrid[i,j];
    barrier(i);
  }
}
```

Figure 3.19 Grid computation for solving Laplace's equation.

- Convergence can be checked with partial sums algorithm.
- Unroll into two stages to avoid copying back.
- Use red-black successive relaxation (Chapter 11).
- Partition grid into blocks (on asynchronous machines).

```
int a[n], sum[n];
process Sum[i = 0 to n-1] {
  sum[i] = a[i];   /* initialize elements of sum */
  while (d < n) {
    if ((i-d) >= 0)    /* update sum */
      sum[i] = sum[i-d] + sum[i];
    d = d + d;     /* double the distance */
  }
}
```

Computing partial sums on a SIMD machine.

- **Single Instruction Multiple Data**
- Every processor executes exactly the same instructions in lock step.
- Barriers not needed since all finish before looping.
- Every process fetches old `sum` before writing new one.
- Parallel assignments thus appear to be atomic.
- `if` statements always take the maximum time.

```
while (true) {
  get a task from the bag;
  if (no more tasks)
    break;     # exit the while loop
  execute the task, possibly generating new ones;
}
```

Outline of worker processes using the bag-of-tasks paradigm.

- **Bag of Tasks**
- Can be used with recursive parallelism (calls are tasks).
- Scalable: use any number of processors.
- Automatic load balancing.

```
int nextRow = 0; # the bag of tasks
double a[n,n], b[n,n], c[n,n];

process Worker[w = 1 to P] {
  int row;
  double sum;  # for inner products
  while (true) {
    # get a task
    ⟨ row = nextRow; nextRow++; ⟩
    if (row >= n)
      break;
    compute inner products for c[row,*];
  }
}
```

Figure 3.20 Matrix multiplication using a bag of tasks.

```
type task = (double left, right, fleft, fright, lrarea);
queue bag(task);      # the bag of tasks
int size;             # number of tasks in bag
int idle = 0;         # number of idle workers
double total = 0.0;   # the total area

compute approximate area from a to b;
insert task (a, b, f(a), f(b), area) in the bag;
count = 1;

process Worker[w = 1 to PR] {
  double left, right, fleft, fright, lrarea;
  double mid, fmid, larea, rarea;
  while (true) {
    # check for termination
    ⟨ idle++;
      if (idle == n && size == 0) break; ⟩
    # get a task from the bag
    ⟨ await (size > 0)
      remove a task from the bag;
      size--; 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) in the bag;
        put (mid, right, fmid, fright, rarea) in the bag;
        size = size + 2; ⟩
    } else
      ⟨ total = total + lrarea; ⟩
  }
  if (w == 1)   # worker 1 prints the result
    printf("the total is %f\n", total);
}
```

Typo: remove `if (w == 1)`

Figure 3.21 Adaptive quadrature using a bag of tasks.