Bakery Algorithm

- Also called Lamport's bakery algorithm
  - after Leslie Lamport
  - A New Solution of Dijkstra's Concurrent Programming Problem Communications of the ACM 17, 8 (August 1974), 453-455.
- This is a mutual exclusion algorithm to prevent concurrent threads from entering critical sections concurrently
- source: wikipedia

Bakery Algorithm

- Analogy
  - bakery with a numbering machine
  - each customer receives unique number
    - numbers increase by one as customers enter
  - global counter displays number of customer being served currently
    - all others wait in queue
  - after baker is done serving customer the next number is displayed
  - served customer leaves

Bakery Algorithm

- threads and bakery analogy
  - when thread wants to enter critical section it has to make sure it has the smallest number.
    - however, with threads it may not be true that only one thread gets the same number
      - e.g., if number operation is non-atomic
    - if more that one thread has the smallest number then the thread with lowest id can enter
    - use pair (number, ID)
      - In this context (a,b) < (c,d) is equivalent to
      - (a<c) or ((a==c) and (b<d))

Bakery Algorithm

from wikipedia

```
// declaration and initial values of global variables
Entering: array [1..N] of bool = {false};
Number: array [1..N] of integer = {0};

lock(integer i)
{
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[N]);
    Entering[i] = false;
    for (j = 1; j <= N; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
        while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) {
            /* nothing */
        }
    }
}
unlock(integer i) { Number[i] = 0; }

Thread(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
}
```

Peterson's Algorithm 1981

- solves critical section problem
- based on shared memory for communication

Peterson's Algorithm

from wikipedia

```
flag[0]   = 0
flag[1]   = 0
turn      = 0

P0: flag[0] = 1
    turn = 1
    while( flag[1] && turn == 1 );
            // do nothing
    // critical section
    ...
    // end of critical section
    flag[0] = 0

P1: flag[1] = 1
    turn = 0
    while( flag[0] && turn == 0 );
            // do nothing
    // critical section
    ...
    // end of critical section
    flag[1] = 0
```

flag value 1 means process wants to enter critical section

Semaphores

- Special variable called a semaphore is used for signaling
- If a process is waiting for a signal, it is suspended until that signal is sent

Semaphores

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - ***Wait*** operation decrements the semaphore value
  - ***Signal*** operation increments semaphore value

Semaphore Primitives

```
struct semaphore {
     int count;
     queueType queue;
}

void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0)
     {
          place this process in s.queue;
          block this process
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0)
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Binary Semaphore Primitives

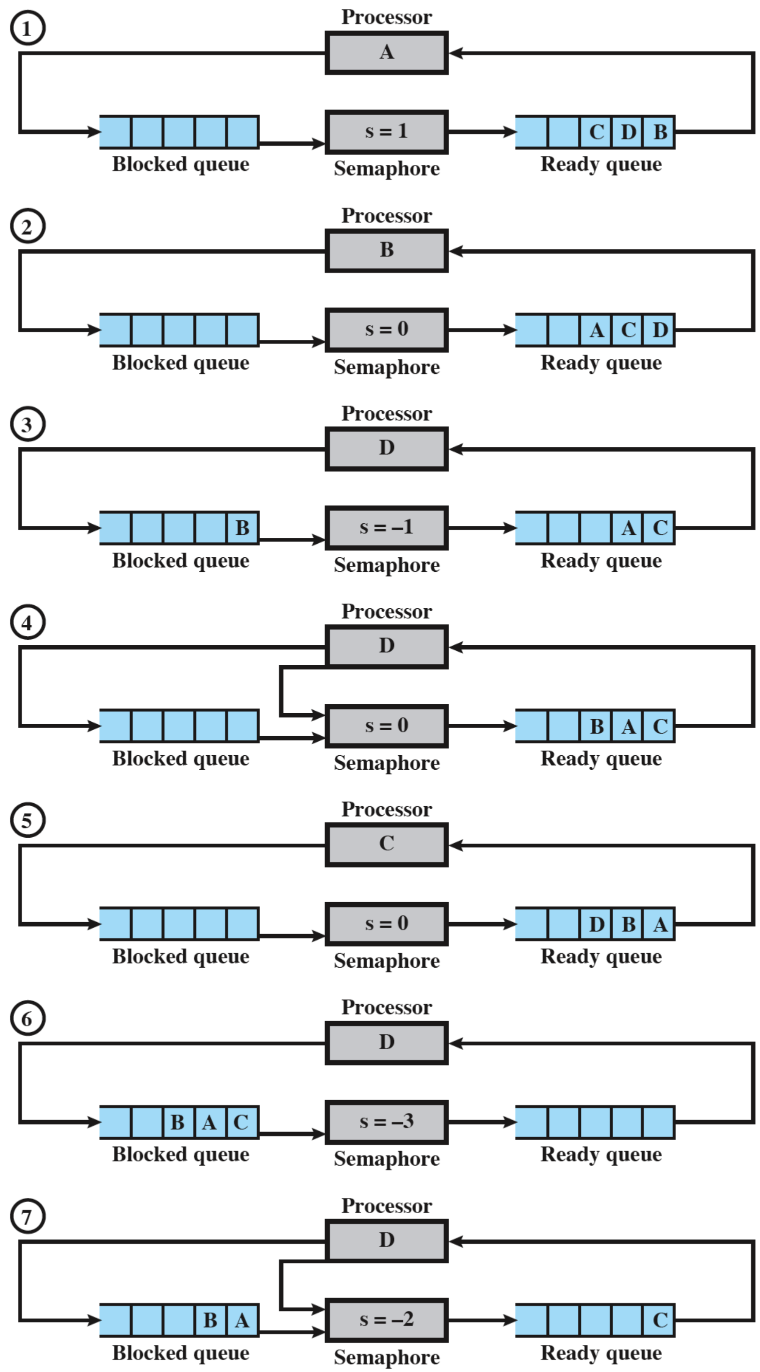
```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Assume process A,B and C depend on result of process D

Initially one result of D is available (s = 1)

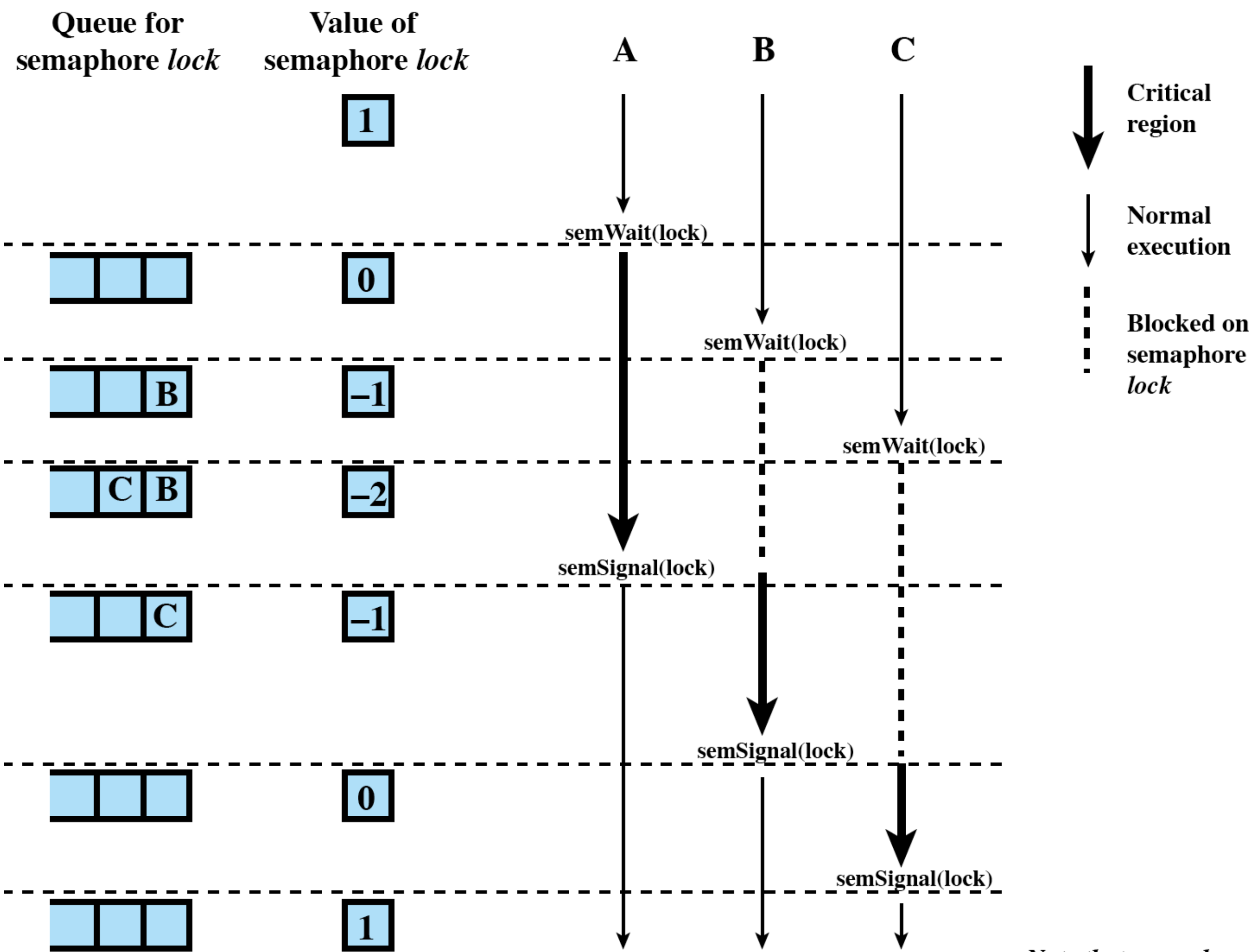


Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true)
     {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

Assume 3 processes, A, B and C



Note that normal execution can proceed in parallel but that critical regions are serialized.