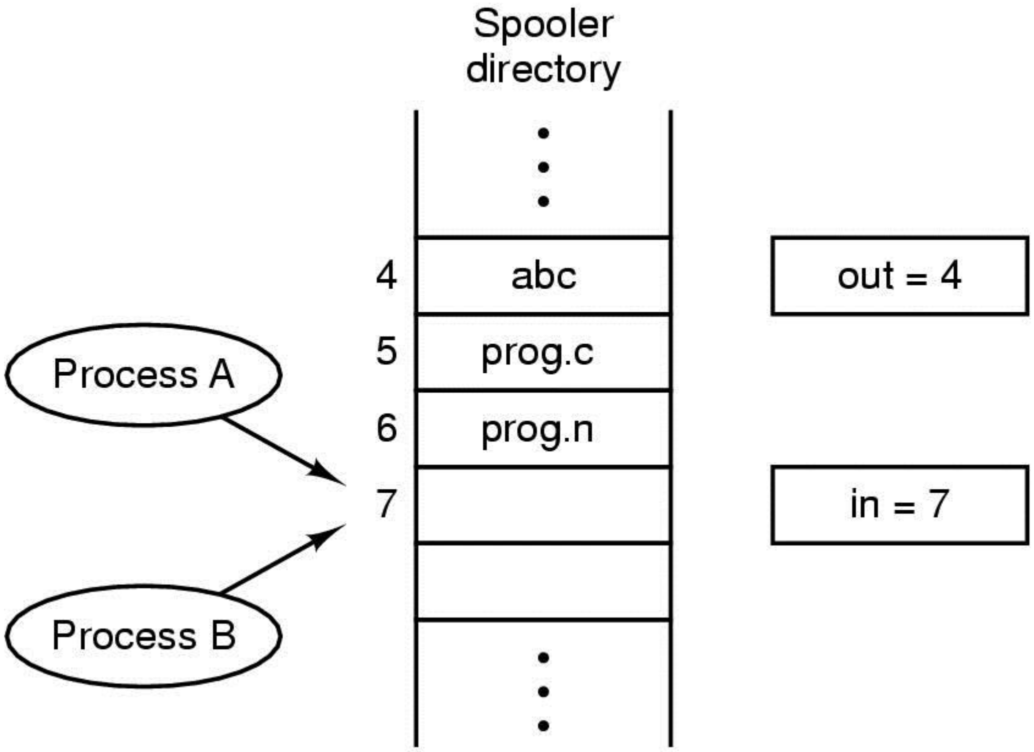
Interprocess Communication

- Three issues
  - How one process can pass information to another
  - Making sure two or more processes do not get into each other's way when engaging in critical activities
  - Proper sequencing when dependencies are present
    - If process A produces data and process B prints them, B has to wait until A has produced some data before starting to print.

Race Conditions

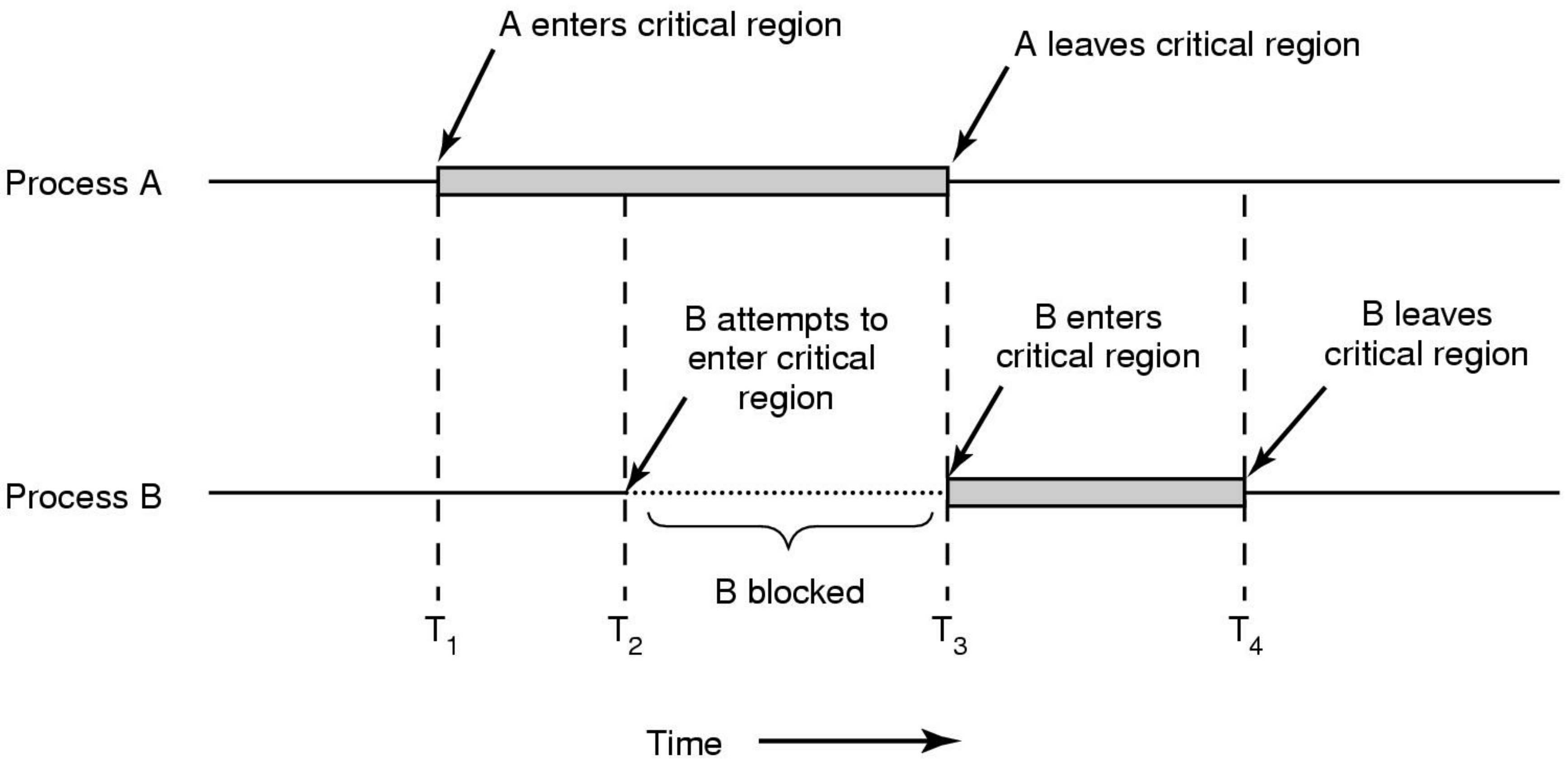
- Race conditions
  - Situation where two or more processes are reading or writing some shared data at the same time and the final result depends on who runs precisely when



Critical Regions (1)

- Mutual Exclusion
  - If one process is using a shared variable or file, the other processes will be excluded from doing the same thing.
- Critical Region (Critical Section)
  - A part of a program where the shared memory is accessed
- Four conditions of a good solution to critical-section problem
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block other processes
  - No process must wait forever to enter its critical region

Critical Regions (2)



Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting (0)

- Disabling Interrupts
  - Disable all interrupts just after entering its critical region and re-enable them just before leaving it
  - Unwise to give user processes to power to turn off interrupts
  - It's convenient for the kernel to disable interrupts while it is updating variables or lists(e.g. list of ready processes) in order to prevent an inconsistent state.
- Lock variables
  - Software solution
  - Use of shared(lock) variable
    - When a process wants to enter its critical region, it first tests the lock.
    - If the lock is 0, the process sets it to 1 and enters the critical region.
    - If the lock is already 1, the process just waits until it becomes 0.
  - The same flaw as the spooler directory

Mutual Exclusion with Busy Waiting (1)

- Strict Alteration solution to critical region problem
  - Busy waiting
    - Continuously testing a variable until some value appears
    - Needs to be avoided, since it wastes CPU time
    - Used only when the waiting time is expected to be very short.
  - Spin lock: a lock that uses busy waiting
  - Avoids all races
  - Violates condition 3
    - A process is blocked by the other not in its critical region
  - Taking turns is not good when one of the processes is much slower than the other.
  - Requires that two processes strictly alternate in entering their critical regions
    - E.g. In spooling files, neither process would be permitted to spool two in a row. → problem

```
while (TRUE) {
    while (turn != 0)       /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)       /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Mutual Exclusion with Busy Waiting (2)

- Peterson's solution for achieving mutual exclusion
  - Much simpler solution than Dekker's solution that does not require strict alteration

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */

void enter_region(int process);     /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 – process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)      /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

Mutual Exclusion with Busy Waiting (2)

- Peterson's solution for achieving mutual exclusion
  - The process calling enter_region enters the critical region(passes through while loop)
    - If the other process is not interested or
    - If the other process is interested and already set the turn
  - E.g. Both processes call enter_region almost simultaneously
    - Both will store their process number in turn. Whichever store is done last is the one that counts. Suppose that process 1 stores last, so turn is 1.
    - When both processes come to the while statement, process 0 executes it zero times and enters its critical region.
    - Process 1 loops and does not enter its critical region until process 0 exits its critical region.

Mutual Exclusion with Busy Waiting (3)

- TSL instruction
  - TSL RX, LOCK (Test and Set Lock)
  - Reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*
  - The operation is guaranteed to be indivisible.
  - Now, we can use a shared variable, *lock*, to coordinate access to shared memory with the help of TSL instruction.

```
enter_region:
    TSL REGISTER,LOCK           | copy lock to register and set lock to 1
    CMP REGISTER,#0             | was lock zero?
    JNE enter_region            | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0                | store a 0 in lock
    RET | return to caller
```

Sleep and Wakeup

- Problems of busy waiting
  - Wasting of CPU time
  - Priority inversion problem

- Sleep and Wakeup
  - Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up by calling a wakeup system call

Sleep and Wakeup

- Producer-consumer problem
  - Also known as the bounded-buffer problem
  - Two processes share a common, fixed-size buffer
    - producer
      - Puts information into the buffer
    - consumer
      - Takes the information out of the buffer
  - If the producer wants to put a new item in the buffer, but the buffer is already full, it goes to sleep until the consumer removes an item from the buffer and wakes it up.
  - Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

Sleep and Wakeup

- Producer-consumer problem
  - Race condition occurs when a wakeup set to a process that is not(yet) sleeping is lost.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count − 1;                    /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Semaphores

- Semaphore
  - New variable type introduced by E. W. Dijkstra in 1965
- Usage

down(P) operation

critical region

up(V) operation

Semaphores

- down(P) operation (P for Dutch proberen(test))
  - If a semaphore is greater than 0, it decrements the value and just continues.
  - If the semaphore is 0, the process is put to sleep without completing the down for the moment.
- up(V) operation (V for verhogen(increment))
  - If one or more processes were sleeping on that semaphore, unable to complete an earlier operation, one of them is awakened and allowed to complete its down.
  - Otherwise, increments the value of the semaphore addressed.
- Each operation above is indivisible atomic action.
  - Normally implemented as system calls where all interrupts are briefly disabled

Semaphores

- Solving the Producer-Consumer Problem using Semaphores
  - Semaphores solve the lost-wakeup problem
  - For mutual exclusion
    - Guarantees that only one process at a time reads or writes the buffer and the associated variable
    - *mutex*
      - Makes sure the producer and consumer do not access the buffer at the same time
      - binary semaphore : initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time
  - For synchronization
    - Guarantees that the producer stops running when the buffer is full, and the consumer stops running when it is empty
    - *full*
      - Counts the number of slots that are full
    - *empty*
      - Counts the number of slots that are empty

Semaphores

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                    /* TRUE is the constant 1 */
          item = produce_item( );      /* generate something to put in buffer */
          down(&empty);                /* decrement empty count */
          down(&mutex);                /* enter critical region */
          insert_item(item);           /* put new item in buffer */
          up(&mutex);                  /* leave critical region */
          up(&full);                   /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                    /* infinite loop */
          down(&full);                 /* decrement full count */
          down(&mutex);                /* enter critical region */
          item = remove_item( );       /* take item from buffer */
          up(&mutex);                  /* leave critical region */
          up(&empty);                  /* increment count of empty slots */
          consume_item(item);          /* do something with the item */
     }
}
```

The producer-consumer problem using semaphores

Mutexes

- Simplified version of the semaphore
- A variable that can be in one of two states:
  - Unlocked : 0
  - Locked : non-zero
- Usage

mutex_lock

critical_region

mutex_unlock

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX              | copy mutex to register and set mutex to 1
        CMP REGISTER,#0                 | was mutex zero?
        JZE ok                          | if it was zero, mutex was unlocked, so return
        CALL thread_yield               | mutex is busy; schedule another thread
        JMP mutex_lock                  | try again later
ok:     RET | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0                   | store a 0 in mutex
        RET | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*

(compare it with the previous example on TSL)

Sharing data between processes

- The shared data structures, such as the semaphores, can be stored in the kernel and only accessed via system calls.

- Most modern operating systems(including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes.

- If nothing else is possible, a shared file can be used.

Monitors

- Problem with semaphores
  - Hard to use
- Monitor
  - High-level synchronization primitive
  - Collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package
  - Processes cannot directly access the monitor's internal data structures.
- Mutual exclusion
  - Only one process can be active in a monitor at any instant.
- Programming language construct
  - Since the compiler arranges for mutual exclusion, it is much less likely that something will go wrong.

Monitors

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

Example of a monitor

Monitors

- Condition variables with two operations:
  - *wait on a condition variable*
    - Causes the calling process to block on a condition variable
    - Allows another process to enter the monitor
  - *signal on a condition variable*
    - Wakes up the process sleeping on the condition variable
  - Used with mutual exclusion guarantee
  - Signals are not accumulated.
    - If a condition variable is signaled with no one waiting on it, the signal is lost forever.
- Problems
  - Should be supported by the compiler
  - Inapplicable to distributed environment (multiple CPUs, each with its own priviate memory, connected by a local network)

Monitors

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count − 1;
                if count = N − 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has *N* slots

Monitors

```
public class ProducerConsumer {
      static final int N = 100;                    // constant giving the buffer size
      static producer p = new producer( );   // instantiate a new producer thread
      static consumer c = new consumer( );// instantiate a new consumer thread
      static our_monitor mon = new our_monitor( );  // instantiate a new monitor
      public static void main(String args[ ]) {
         p.start( );                              // start the producer thread
         c.start( );                              // start the consumer thread
      }
      static class producer extends Thread {
         public void run( )  {                    // run method contains the thread code
            int item;
            while (true) {                        // producer loop
               item = produce_item( );
               mon.insert(item);
            }
         }
         private int produce_item( ) { ... }     // actually produce
      }
      static class consumer extends Thread {
         public void run( )  {                    run method contains the thread code
            int item;
            while (true) {                        // consumer loop
               item = mon.remove( );
               consume_item (item);
            }
         }
         private void consume_item(int item) { ... }     // actually consume
      }
```

Solution to producer-consumer problem in Java (part 1)

Monitors

```
static class our_monitor {                        // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );   // if the buffer is full, go to sleep
        buffer [hi] = val;                // insert an item into the buffer
        hi = (hi + 1) % N;                // slot to place next item in
        count = count + 1;                // one more item in the buffer now
        if (count == 1) notify( );        // if consumer was sleeping, wake it up
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );   // if the buffer is empty, go to sleep
        val = buffer [lo];                // fetch an item from the buffer
        lo = (lo + 1) % N;                // slot to fetch next item from
        count = count – 1;                // one few items in the buffer
        if (count == N – 1) notify( );    // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
  }
}
```

Solution to producer-consumer problem in Java (part 2)

Message Passing

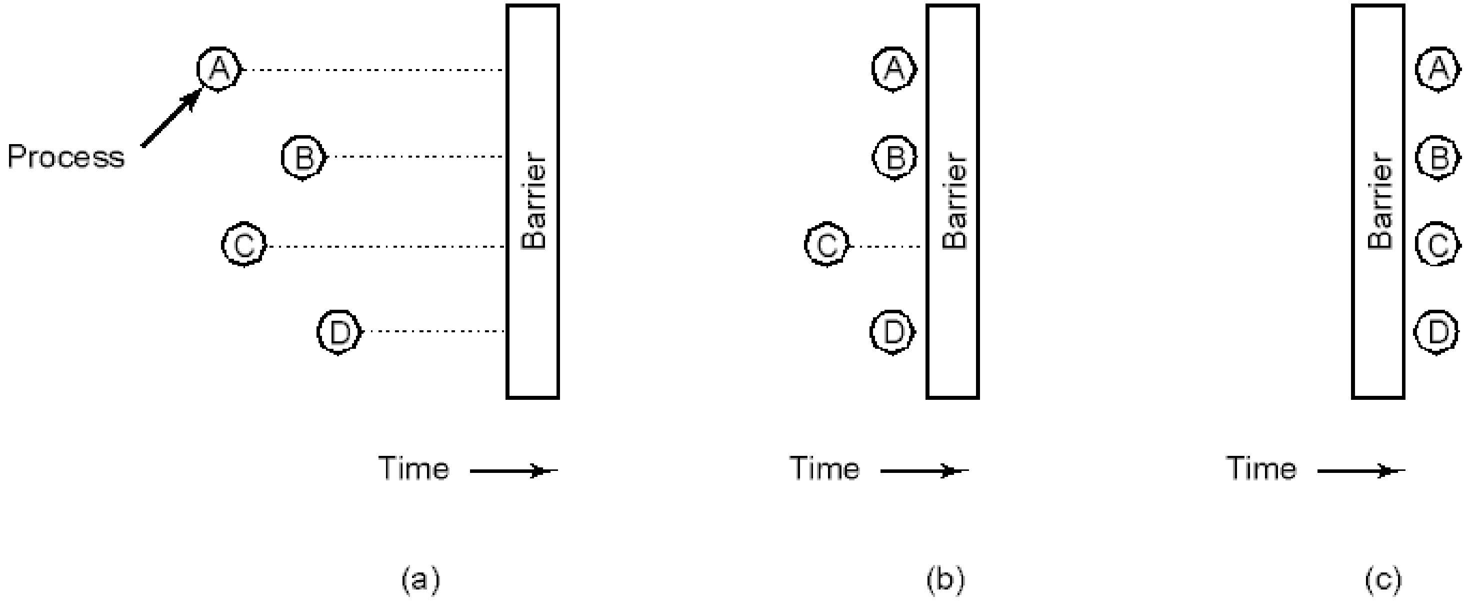
- Method of interprocess communication
  - Uses send and receive system calls
  - send(destination, &message)
    - Sends a message to a given destination
  - receive(source, &message)
    - Receives a message from a given source
    - If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.
- Design issues
  - Messages can be lost by the network.
    - Acknowledement, sequence number
  - How processes are named
  - Authentication
  - Performance issue when the sender and receiver are on the same machine

Message Passing

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
      int item;
      message m;                                /* message buffer */

      while (TRUE) {
            item = produce_item( );             /* generate something to put in buffer */
            receive(consumer, &m);              /* wait for an empty to arrive */
            build_message(&m, item);            /* construct a message to send */
            send(consumer, &m);                 /* send item to consumer */
      }
}

void consumer(void)
{
      int item, i;
      message m;

      for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
      while (TRUE) {
            receive(producer, &m);              /* get message containing item */
            item = extract_item(&m);            /* extract item from message */
            send(producer, &m);                 /* send back empty reply */
            consume_item(item);                 /* do something with the item */
      }
}
```

The producer-consumer problem with N messages

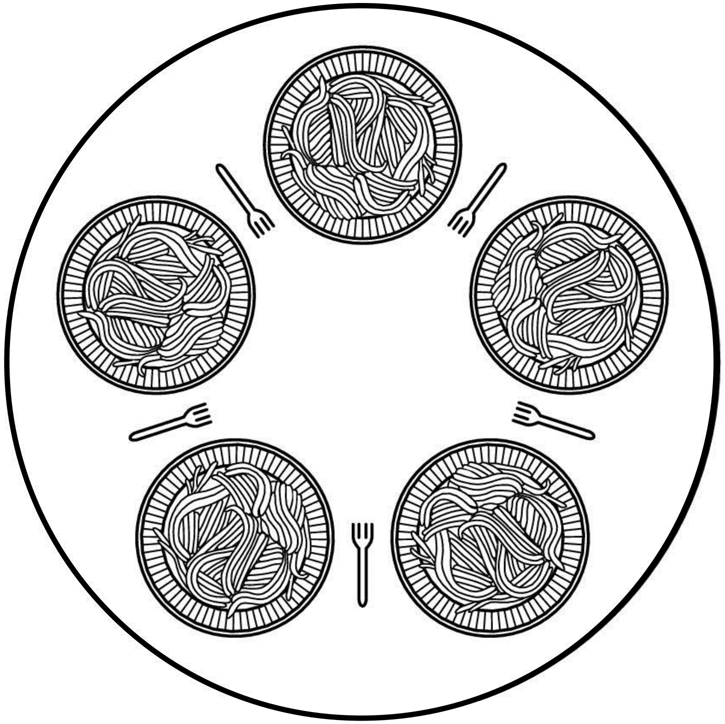
Barriers



- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through

Dining Philosophers

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



Dining Philosophers

- A wrong solution to the dining philosophers problem
  - causes deadlock

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```

Dining Philosophers

- 2nd solution
  - After taking the left fork, check to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process.
  - Could cause starvation
- 3rd solution
  - Philosophers wait a random time instead of the same time after failing to acquire the right-hand fork.
- 4th solution
  - Protect the five statements following *think* by a binary semaphore
  - Theoretically, this is adequate.
  - But, practically, it has a performance bug: only one philosopher can be eating at any instant.

Dining Philosophers

```
#define N            5              /* number of philosophers */
#define LEFT         (i+N−1)%N      /* number of i's left neighbor */
#define RIGHT        (i+1)%N        /* number of i's right neighbor */
#define THINKING     0              /* philosopher is thinking */
#define HUNGRY       1              /* philosopher is trying to get forks */
#define EATING       2              /* philosopher is eating */
typedef int semaphore;              /* semaphores are a special kind of int */
int state[N];                       /* array to keep track of everyone's state */
semaphore mutex = 1;                /* mutual exclusion for critical regions */
semaphore s[N];                     /* one semaphore per philosopher */

void philosopher(int i)             /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                  /* repeat forever */
        think( );                   /* philosopher is thinking */
        take_forks(i);              /* acquire two forks or block */
        eat( );                     /* yum-yum, spaghetti */
        put_forks(i);               /* put both forks back on table */
    }
}
```

Solution to dining philosophers problem (part 1)
: deadlock-free and allows the maximum parallelism

Dining Philosophers

```
void take_forks(int i)              /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = HUNGRY;            /* record fact that philosopher i is hungry */
      test(i);                      /* try to acquire 2 forks */
      up(&mutex);                   /* exit critical region */
      down(&s[i]);                  /* block if forks were not acquired */
}

void put_forks(i)                   /* i: philosopher number, from 0 to N–1 */
{
      down(&mutex);                 /* enter critical region */
      state[i] = THINKING;          /* philosopher has finished eating */
      test(LEFT);                   /* see if left neighbor can now eat */
      test(RIGHT);                  /* see if right neighbor can now eat */
      up(&mutex);                   /* exit critical region */
}

void test(i)                        /* i: philosopher number, from 0 to N–1 */
{
      if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
            state[i] = EATING;
            up(&s[i]);
      }
}
```

Solution to dining philosophers problem (part 2)