6 Synchronization

Contents

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors

Reentrant Code

- A function/method is said to be "reentrant" if…

  "A function that has been invoked may be invoked again before the first invocation has returned, and will still work correctly."

- In the context of multiprogramming…

  A reentrant function can be executed simultaneously by more than one thread, with no ill effects.

Reentrant Code (cont'd)

- Consider this function:

```
int count;

int GetUnique()
{
    count = count + 1;
    return count
}
```

- What if it is executed by different threads?
  - The results may be incorrect!
  - This routine is not reentrant!

Reentrant Code (cont'd)

- When is code "reentrant"?
- Assumption:
  - A multi-threaded program
  - Some variables are
    - "local" -- to the function/method/routine
    - "global" -- sometimes called "static"
- Access to local variables?
  - A new stack frame is created for each invocation.
- Access to global variables?
  - Must use synchronization!

Reentrant Code (cont'd)

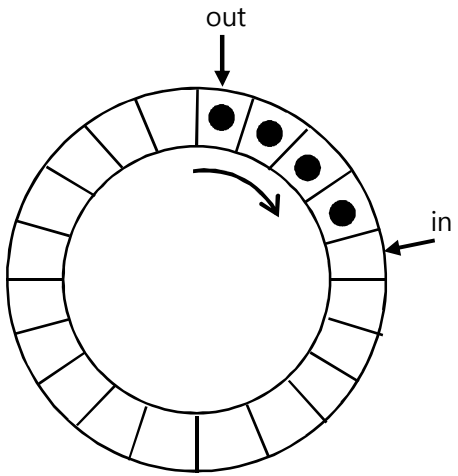
- Making the function reentrant

```
int count;
Mutex lock;

int GetUnique()
{
    int i;

    lock(mylock);
    count = count + 1;
    i = count;
    unlock(mylock);

    return i;
}
```

Producer vs. Consumer

Producer

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```



buffer[BUFFER_SIZE]

Consumer

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

Race Condition

- counter++ could be implemented as

$$register_1 = \texttt{counter}$$
$$register_1 = register_1 + 1$$
$$\texttt{counter} = register_1$$

- counter-- could be implemented as

$$register_2 = \texttt{counter}$$
$$register_2 = register_2 - 1$$
$$\texttt{counter} = register_2$$

- Consider this execution interleaving with "counter = 5" initially:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

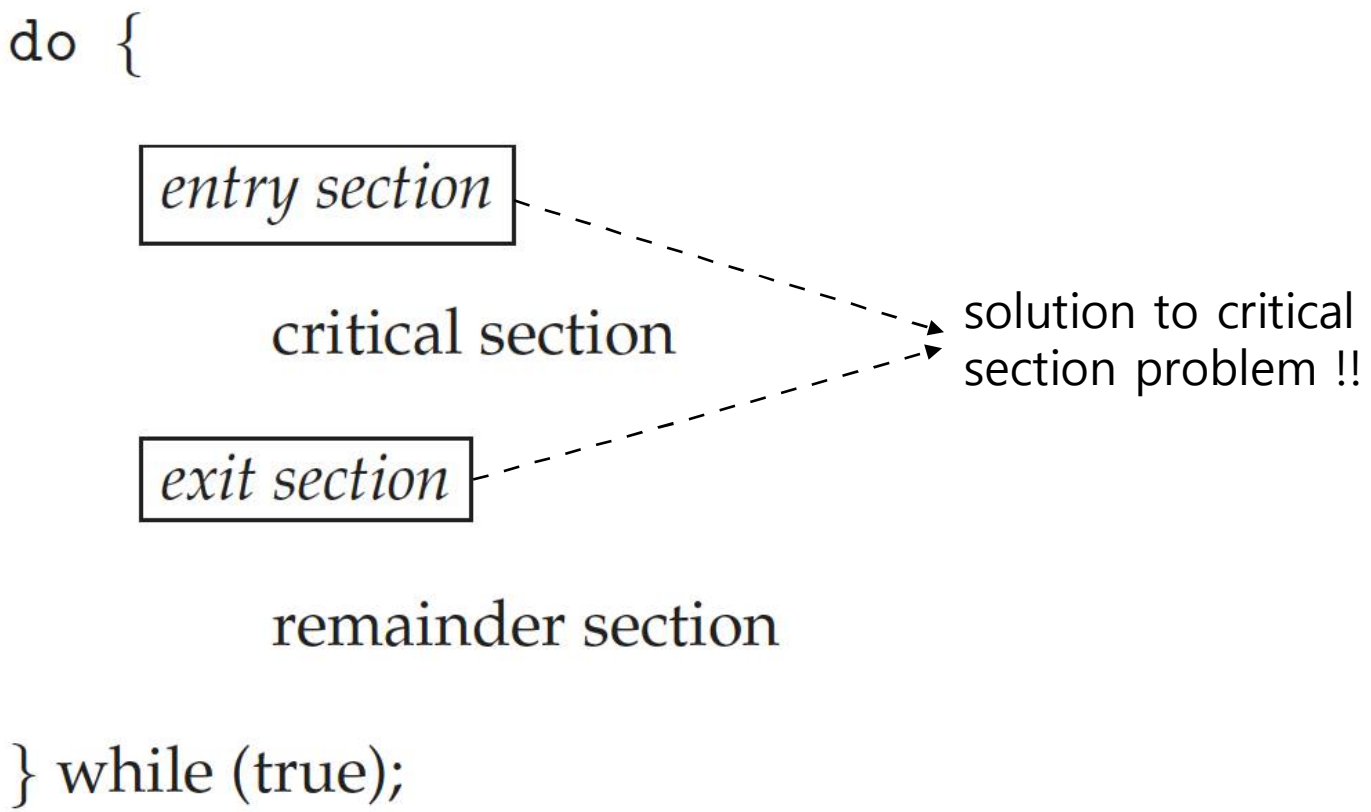
Terms

- Asynchronous concurrent processes
- Race condition
- Critical section
- Mutual exclusion

Solution to Critical Section Problem

- A solution to the critical section problem must satisfy the following requirements
  - Mutual Exclusion : If a process is executing in its critical section, then no other processes can be executing in their critical sections
  - Progress : If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
  - Bounded Waiting : A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the processes

General Structure of a Typical Process P_i in a Solution to the Critical Section Problem



Peterson's Solution

- S/W-based two process solution
- Assume that the `LOAD` and `STORE` instructions are atomic; that is, they cannot be interrupted.
- The two processes, P_i and P_j, share two variables:
  - `int turn;`
  - `boolean flag[2];`
- The variable `turn` indicates whose turn it is to enter the critical section.
- The `flag[]` array is used to indicate if a process is ready to enter the critical section. "`flag[i] = true`" implies that process P_i is ready

Peterson's Solution (cont'd)

```
/* Pi */
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

```
/* Pj */
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn == i);

        critical section

    flag[j] = false;

        remainder section

} while (true);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor system; Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions (atomic ≈ non-interruptible)
  - Either test memory word and set value
  - Or swap contents of two memory words

`test_and_set` Instruction

- Definition:

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Solution using `test_and_set`

- Shared boolean variable `lock`, initialized to `false`

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

`compare_and_swap` Instruction

- Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Solution using `compare_and_swap`

- Shared Boolean variable `lock` initialized to `FALSE`

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with `test_and_set`

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

```
boolean waiting[n];
boolean lock;
```

Mutex Locks

- The hardware-based solutions are complicated as well as generally inaccessible to application programmers.
- Instead, OS designers build software tools to solve the critical-section problem. The simplest of these tools is the `mutex` lock. (mutex is short for mutual exclusion.)
- We use mutex lock to prevent race conditions. A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The `acquire()` function acquires the lock and the `release()` function releases the lock

Solution to the Critical Section Problem using Mutex Lock

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}

release() {
   available = true;
}
```

```
do {

    acquire lock

        critical section

    release lock

        remainder section

} while (true);
```

Spin Lock

- The main disadvantage of the implementation is that it requires *busy waiting*
- In fact, this type of mutex lock is also called a *spinlock*. This continual looping is a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.
- Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- Thus, when locks are expected to be held for short times, spinlocks are useful. They are often employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.

Semaphore

- Synchronization tool that does not require busy waiting
  - Semaphore `S` – integer variable
  - Two standard operations modify `S` : `wait()` and `signal()` (`wait()` was originally called `P()`, from the Dutch *proberen*, "to test"; `signal()` was originally called `V()` from *verhogen*, "to increment")
  - Can only be accessed via two indivisible (atomic) operations

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement; also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore S;    //  initialized to 1

wait (S);
Critical Section
signal (S);
```

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

- Note that in this implementation, semaphore values may be negative and its magnitude is the number of processes waiting on that semaphore.
- Two operations on process
  - block : place the process invoking the operation on the waiting queue and the state is switched to waiting
  - wakeup : remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation with no Busy waiting (Cont'd)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let `S` and `Q` be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

Starvation

- Indefinite blocking
- A process may never be removed from the semaphore queue in which it is suspended

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- `N` buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value `N`

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

The Structure of the Producer Process

```
do {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
} while (true);
```

The Structure of the Consumer Process

```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers: only read the data set; do not perform any update
  - Writers: can both read and write.
- Problem – allow multiple readers to read at the same time. Only single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1. It functions as a mutual exclusion semaphore for the writers
  - Integer `read_count` initialized to 0. It keeps track of how many processes are currently reading the object
  - Semaphore `mutex` initialized to 1. to ensure mutual exclusion when the variable read count is updated

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

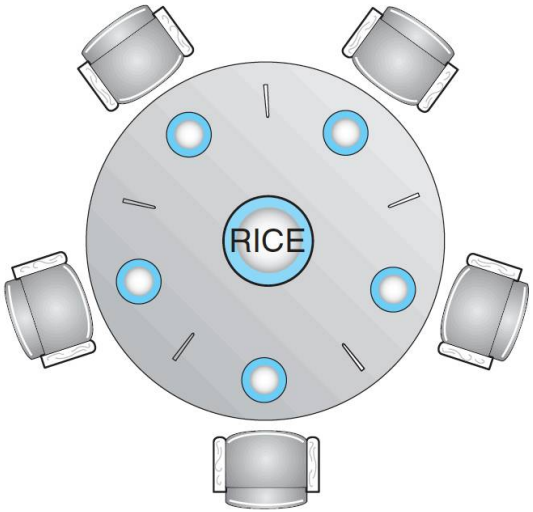
The Structure of a Writer Process

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

The Structure of a Reader Process

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore `chopstick[5]` initialized to 1

The Structure of Philosopher `i`

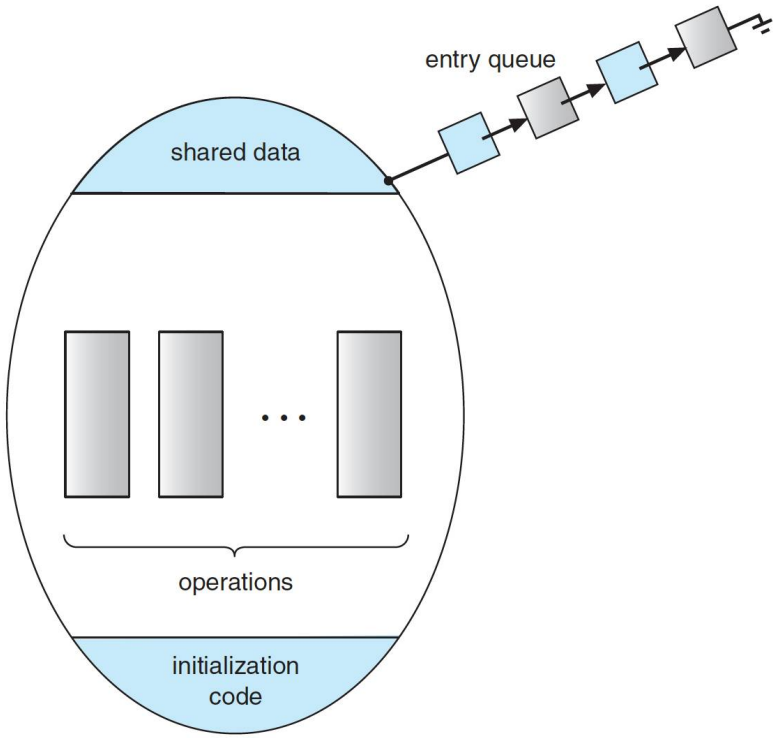
```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Problems with Semaphores

- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time



Syntax of a Monitor

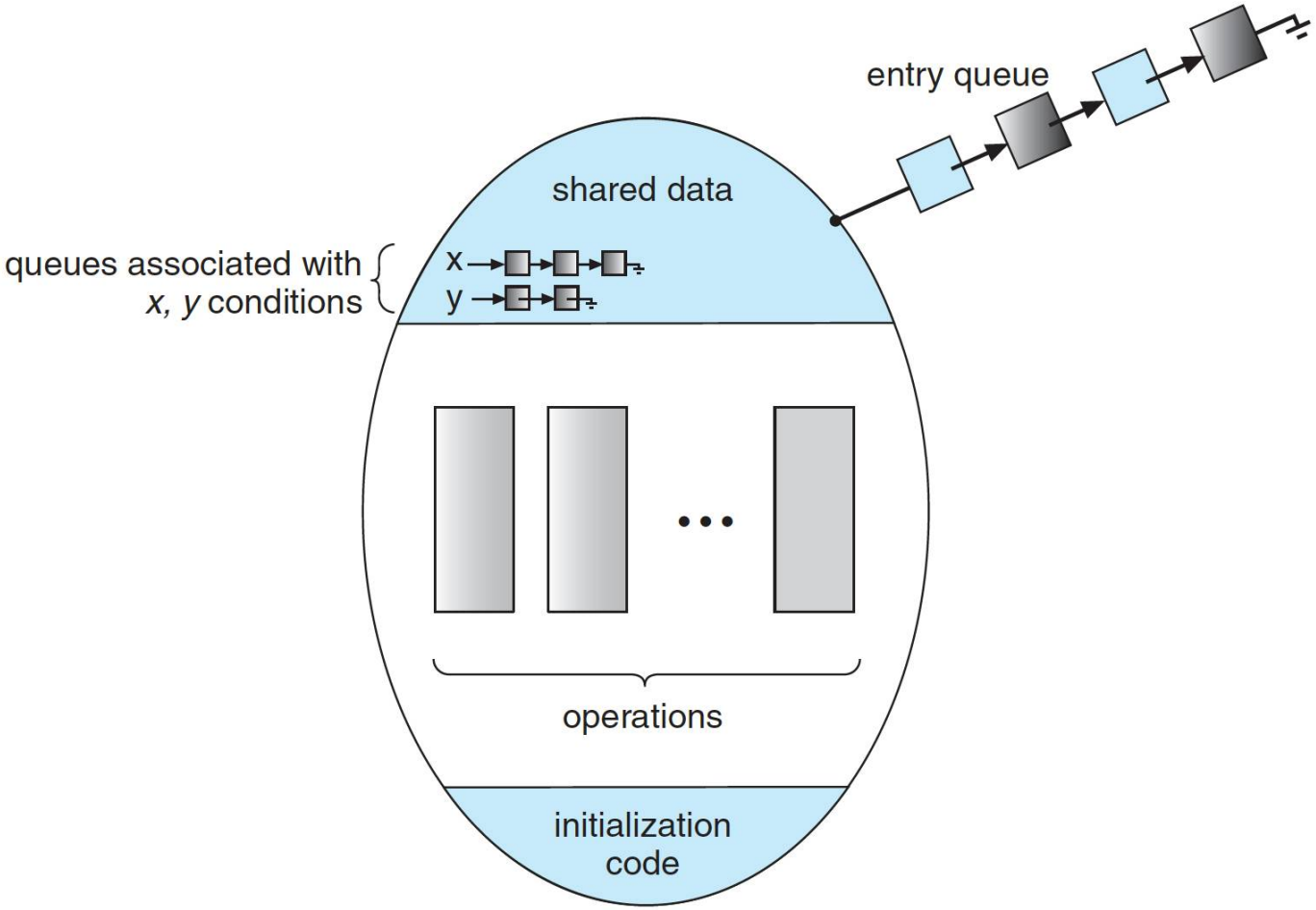
```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
          state[i] = EATING;
          self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

```
DiningPhilosophers.pickup(i);
                ...
                eat
                ...
DiningPhilosophers.putdown(i);
```