CSE2008: Operating Systems

L15 L16 L17 & L18: Process Synchronization

Dr. Subrata Tikadar
SCOPE, VIT-AP University

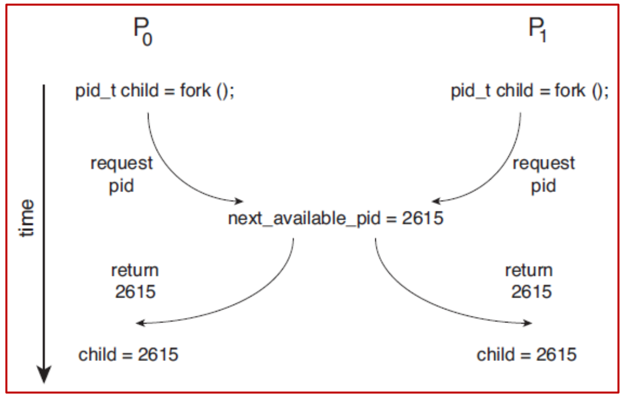
Recap

- Introductory Concepts
- Process Fundamentals
- IPC
- CPU Scheduling Algorithms
- Multithreading Concepts

Outline

- Process synchronization
- Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Monitors
- Classic problems of Synchronization

Process synchronization



- Preliminary Concept
  - Producer–Consumer Problem

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

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

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

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

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

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

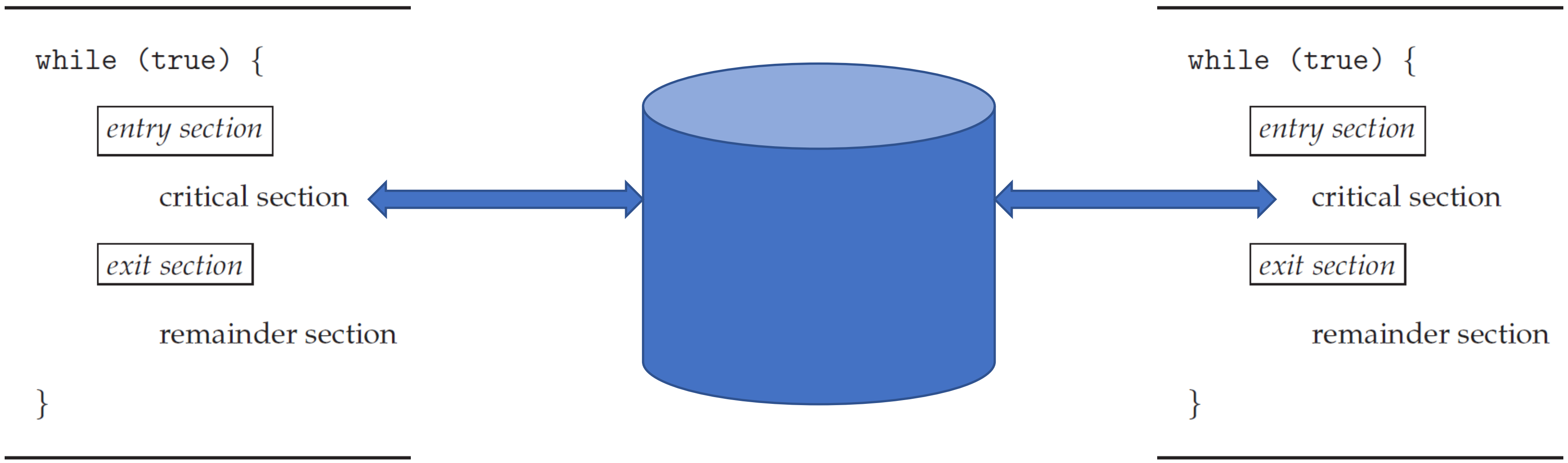
$register_1 = \texttt{count}$
$register_1 = register_1 + 1$
$\texttt{count} = register_1$

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

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

Critical-Section Problem

- Critical-Section Problem
  - Critical-Section: A segment of code in which the process may be accessing — and updating — data that is shared with at least one other process



```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

Critical-Section Problem

- Critical-Section Problem
  - A solution to the critical-section problem must satisfy the following three requirements:
    1. **Mutual exclusion** → If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
    2. **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
    3. **Bounded waiting** → There exists a bound, or limit, 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.

Peterson's Solution

- Peterson's Solution
  - Restricted to two processes that alternate execution between their critical sections and remainder sections
    - The processes are numbered P_0 and P_1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, j equals 1 – i
  - Requires the two processes to share two data items:

int turn;

boolean flag[2];

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Peterson’s Solution

- Peterson’s Solution
  - To prove that this solution is correct, we need to show that:
    1. Mutual exclusion is preserved,
    2. The progress requirement is satisfied, and
    3. The bounded-waiting requirement is met.

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Peterson’s Solution

- Peterson’s Solution
  - To prove that this solution is correct, we need to show that:
    1. Mutual exclusion is preserved,
    2. The progress requirement is satisfied, and
    3. The bounded-waiting requirement is met.

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Each *Pi* enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes—say, P_j—must have successfully executed the while statement, whereas P_i had to execute at least one additional statement (“turn == j”). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as P_j is in its critical section; as a result, **mutual exclusion** is preserved.

Peterson's Solution

- Peterson's Solution
    - To prove that this solution is correct, we need to show that:
        1. Mutual exclusion is preserved,
        2. The progress requirement is satisfied, and
        3. The bounded-waiting requirement is met.

```
int turn;
boolean flag[2];
```

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

A process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If P_j is not ready to enter the critical section, then flag[j] == false, and P_i can enter its critical section. If P_j has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then P_i will enter the critical section. If turn == j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section. If P_j resets flag[j] to true, it must also set turn to i. Thus, since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (**progress**) after at most one entry by P_j (**bounded waiting**).

Synchronization Hardware

- Memory Barriers
- Hardware Instructions
- Atomic Variables

Synchronization Hardware

- Memory Barriers
  - Memory Model:
    1. Strongly ordered → where a memory modification on one processor is immediately visible to all other processors
    2. Weakly ordered → where modifications to memory on one processor may not be immediately visible to other processors

➢ Instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors – such instructions are known as **memory barriers** or **memory fences**.

Synchronization Hardware

- Hardware Instructions
  - test and set() instruction – TAS

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

   return rv;
}
```

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

      /* critical section */

   lock = false;

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

Synchronization Hardware

- Hardware Instructions
  - compare_and_swap() instruction – CAS

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

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

   return temp;
}
```

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

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Synchronization Hardware

- Hardware Instructions
  - compare_and_swap() instruction – CAS

Self Study: *Analyse how this algorithm satisfies all the critical-section requirements!*

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock,0,1);
  waiting[i] = false;

    /* critical section */

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

  if (j == i)
    lock = 0;
  else
    waiting[j] = false;

    /* remainder section */
}
```

Synchronization Hardware

- Atomic Variables

```
increment(&sequence);
```

where the `increment()` function is implemented using the CAS instruction:

```
void increment(atomic_int *v)
{
   int temp;

   do {
      temp = *v;
   }
   while (temp != compare_and_swap(v, temp, temp+1));
}
```

Mutex Locks

```
while (true) {

        acquire lock

            critical section

        release lock

            remainder section

}
```

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

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

Semaphores

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

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

Semaphores

- Categories of Semaphores
  - Counting semaphore

    The value of it can range over an unrestricted domain
  - Binary semaphore

    The value of a binary semaphore can range only between 0 and 1

Semaphores

- Semaphore Usage
  - Example:
    - Let us assume that two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2
    - Also assume that we require that S_2 be executed only after S_1 has completed

```
S1;
signal(synch);
```

```
wait(synch);
S2;
```

Semaphores

- Semaphore Implementation
  - Modified definitions of the wait() and signal() operations [to avoid busy waiting]

    When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can suspend itself. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

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

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

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

Semaphores

- Pros and Cons
  - Pros – provides a convenient and effective mechanism for process synchronization
  - Cons – incorrect usage can result in timing errors that are difficult to detect
    - Example:

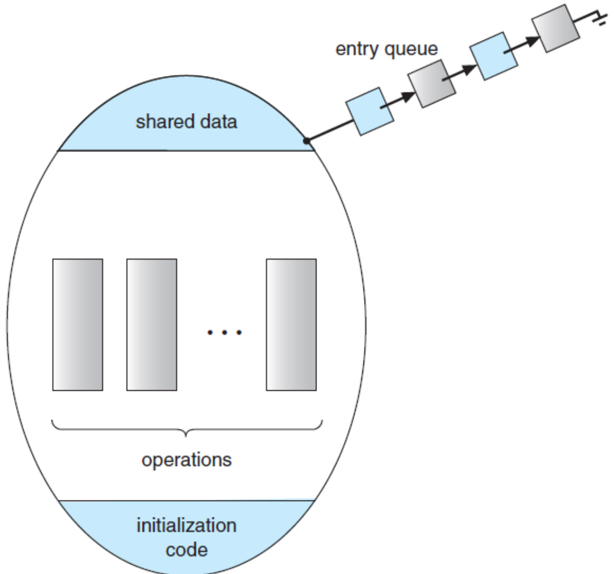
      Suppose that a program interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);          wait(mutex);
   ...                     ...
   critical section        critical section
   ...                     ...
wait(mutex);            wait(mutex);
```

Monitors

- An ADT (abstract data type) that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor
- Also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables

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

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

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

            .
            .
            .

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

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



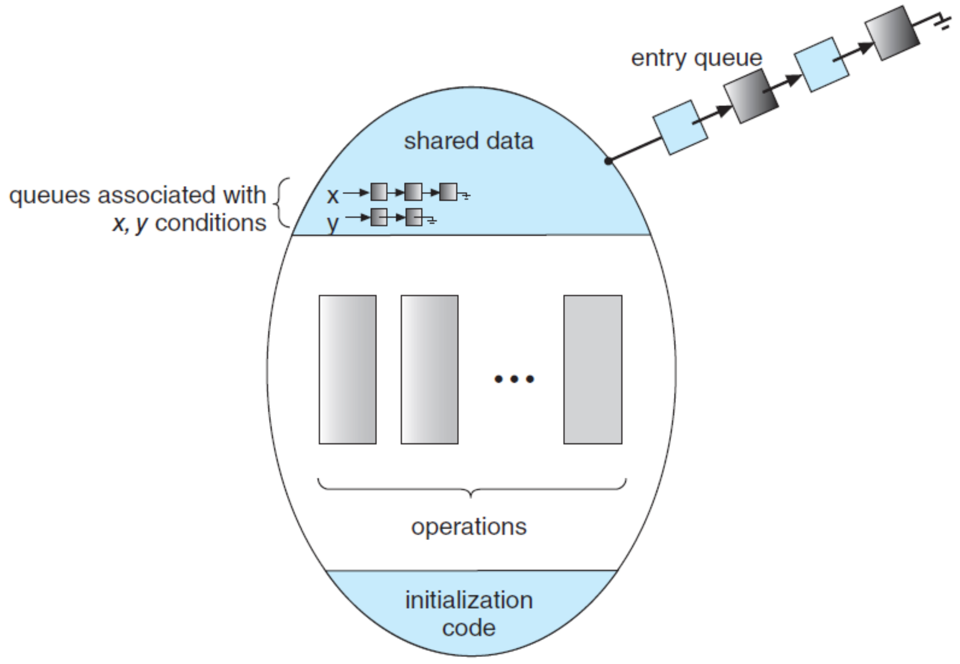
Monitors

- The condition construct
  - A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

    condition x, y;

The only operations that can be invoked on a condition variable are **wait()** *and* **signal()**.

The operation ***x.wait();*** *means that the process invoking this operation is suspended until another process invokes* ***x.signal();***



Monitors

- The condition construct
  - A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

    condition x, y;

The only operations that can be invoked on a condition variable are ***wait()*** *and* ***signal()****.*

The operation ***x.wait();*** *means that the process invoking this operation is suspended until another process invokes* ***x.signal();***

Now suppose that, when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.

→ Two possibilities exist:

1. **Signal and wait.** *P either waits until Q leaves the monitor or waits for another condition.*
2. **Signal and continue.** *Q either waits until P leaves the monitor or waits for another condition.*

Classic problems of Synchronization

- The Bounded-Buffer Problem
- The Readers–Writers Problem
- The Dining-Philosophers Problem

Classic problems of Synchronization

- The Bounded-Buffer Problem
  - Let us assume that the producer and consumer processes share the following data structures:

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

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

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

Classic problems of Synchronization

- The Readers–Writers Problem
  - Readers – the processes that want only to read the database
  - Writers – the processes that want to update the database

If at least one process among the set of concurrent processes is a *writer* then chaos may occur

➢To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database

*This synchronization problem is referred to as **the readers–writers problem**.*

- ***The first readers–writers problem** – No reader be kept waiting unless a writer has already obtained permission to use the shared object*
- ***The second readers–writers problem** – If a writer is waiting to access the object, no new readers may start reading*

Classic problems of Synchronization

- The Readers–Writers Problem
  - Solution to the first readers-writers problem
    - The reader processes share the following data structures:

      semaphore rw_mutex = 1;

      semaphore mutex = 1;

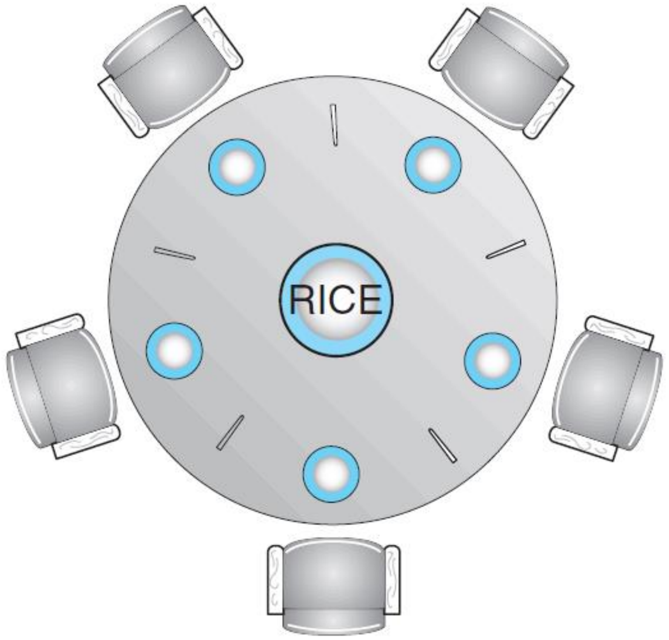
      int read_count = 0;
    - Codes for a writer process and a reader process:

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

```
while (true) {
   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);
}
```

Classic problems of Synchronization

- The Dining-Philosophers Problem



Semaphore Solution

- The shared data:

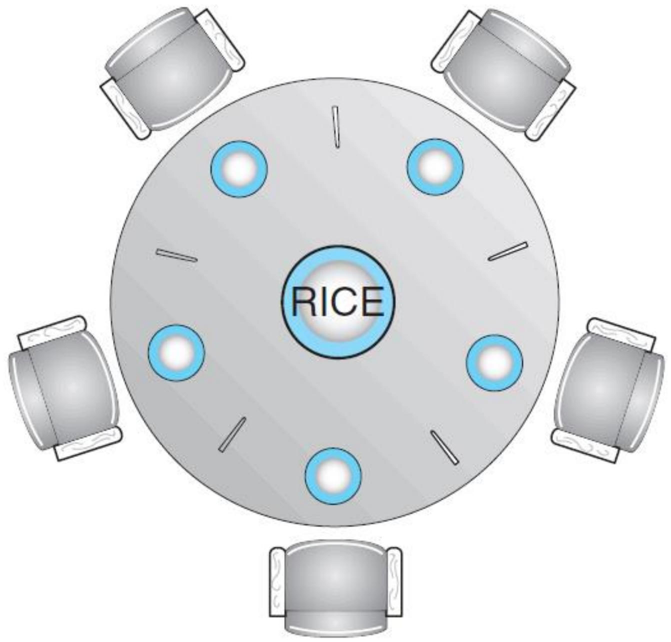
  semaphore chopstick[5];

- The structure of philosopher i:

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

Classic problems of Synchronization

- The Dining-Philosophers Problem



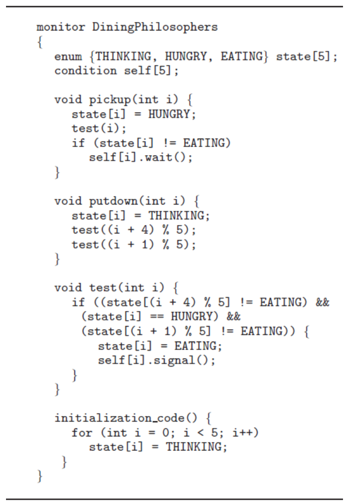
Monitor Solution

- The data structure:

  enum {THINKING, HUNGRY, EATING} state[5];

  condition self[5];

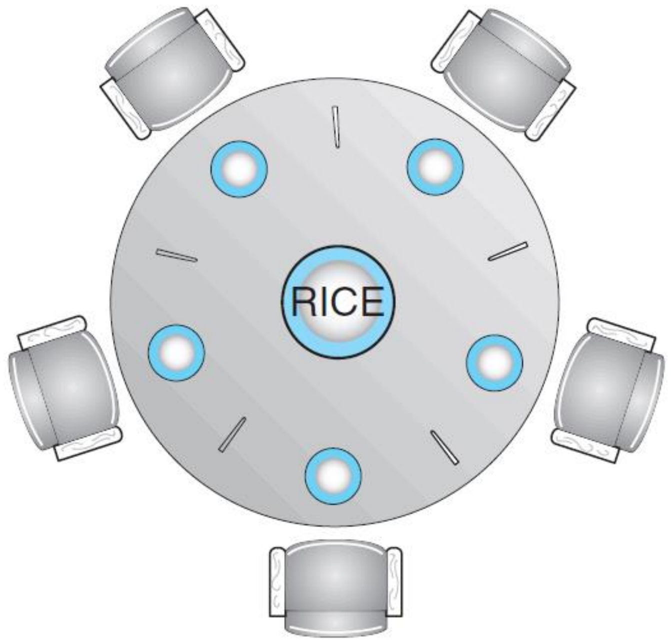
- The structure of philosopher i:



```
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;
    }
}
```

Classic problems of S

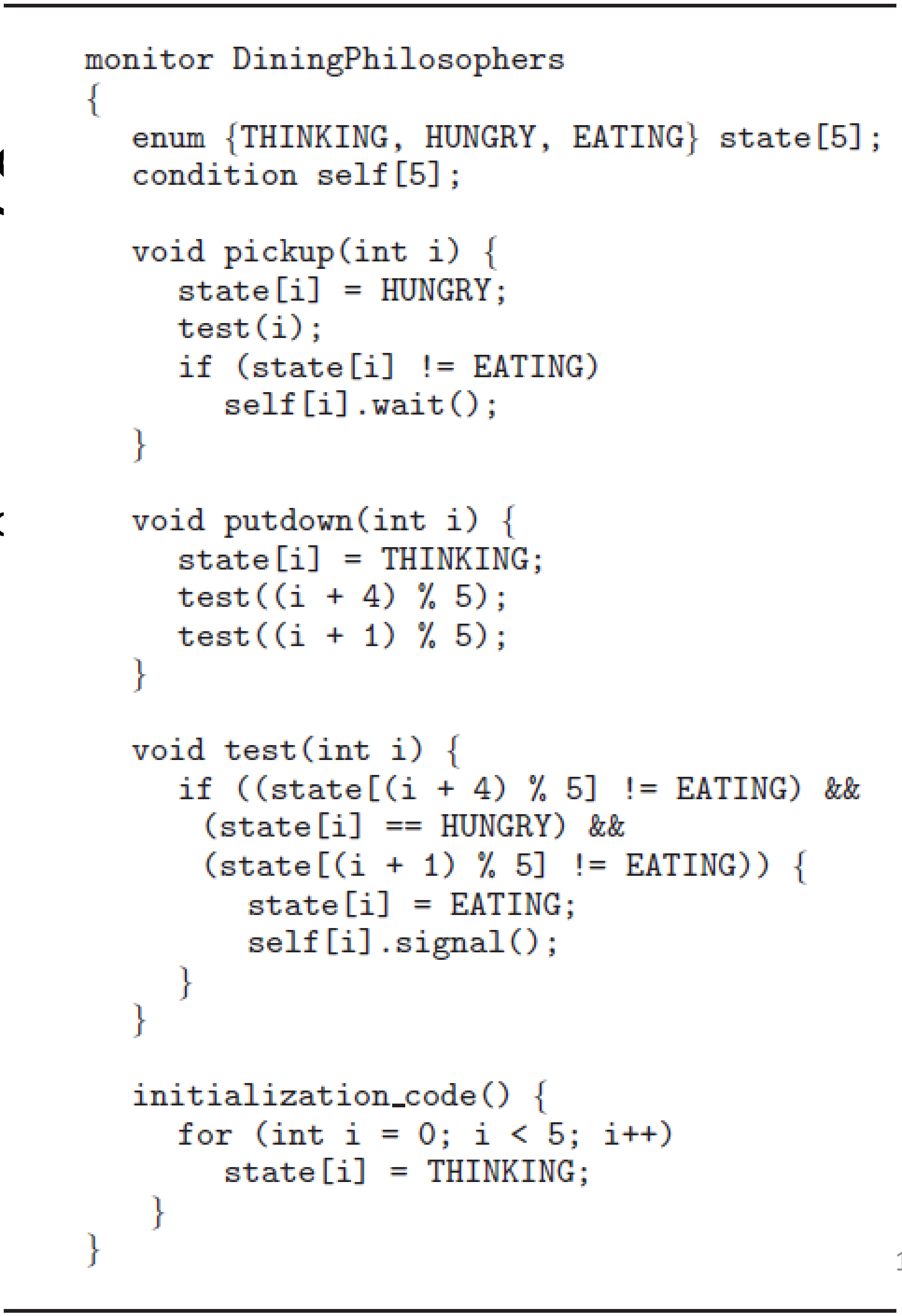
- The Dining-Philosophers Prob



Monito

▪

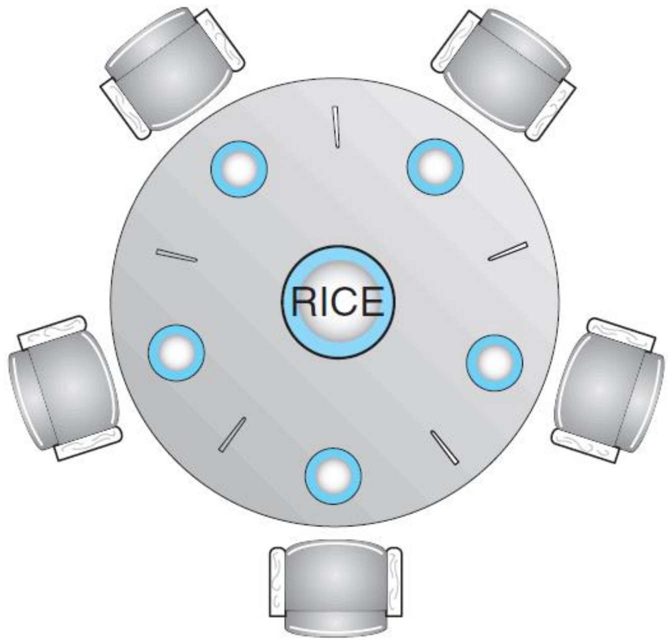
▪



```
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;
    }
}
```

Classic problems of Synchronization

- The Dining-Philosophers Problem



Monitor Solution

- The data structure:

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

- The Philosopher i must invoke the operations pickup() and putdown() in the following sequence:

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

Reference

- Abraham Silberschatz , Peter B. Galvin, Greg Gagne, “Operating System Concepts”, Addison Wesley, 10th edition, 2018
  - Chapter 6: Section 6.1 – 6.7
  - Chapter 7: Section 7.1 – 7.2

Next

- Deadlock

Thank You