University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 12: Concurrency

University of Nottingham, Ningbo China, 2024

Recap
Last Lecture

- The Multiple Consumer, Multiple Producer, Bounded Buffer

- The dining philosophers problem
  - Scenarios
  - Solutions

Today's class

- What are the monitors and how can we use them?
- Monitor Semantics/Structure
- limitation
- Monitor Implementation
- Solving synchronization problems with monitors
  - Pseudocode Example: Monitor for producer consumer Bounded Buffer
  - The dining philosophers problem Solution with Monitor
- Semaphores versus Condition Variables

What's wrong with Semaphores?

- Global Variables: Semaphores are shared variables accessible from anywhere in the program, increasing the risk of misuse.
- No Connection to Data: Lack of inherent linkage between the semaphore and the resource it protects, leading to confusion in complex systems.
- Unrestricted Access: Any part of the code can signal or wait on semaphores, making it hard to enforce correct usage and increasing the chance of errors.
- Dual Purpose Functionality: Serve both mutual exclusion and scheduling, complicating their management and leading to potential misuse.
- No Enforcement of Proper Usage: No built-in checks ensure that semaphores are used correctly, leading to risks like deadlock.

Monitors

- **Deadlock**: Occur when two tasks try to lock two different semaphores in a different order.
  - E.g. if the programmer mistakenly develop this

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

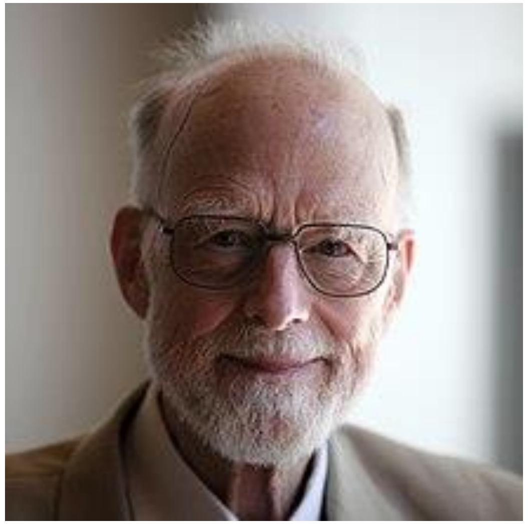
```
wait(mutex);
   ...
   critical section
   ...
wait(mutex);
```

Suppose that a process interchanges the order in which the wait() and signal() operations.

Suppose that a process replaces signal(mutex) with wait(mutex)

- Solution: use a higher level primitive called monitors.

Monitors



Sir Charles Antony Richard Hoare

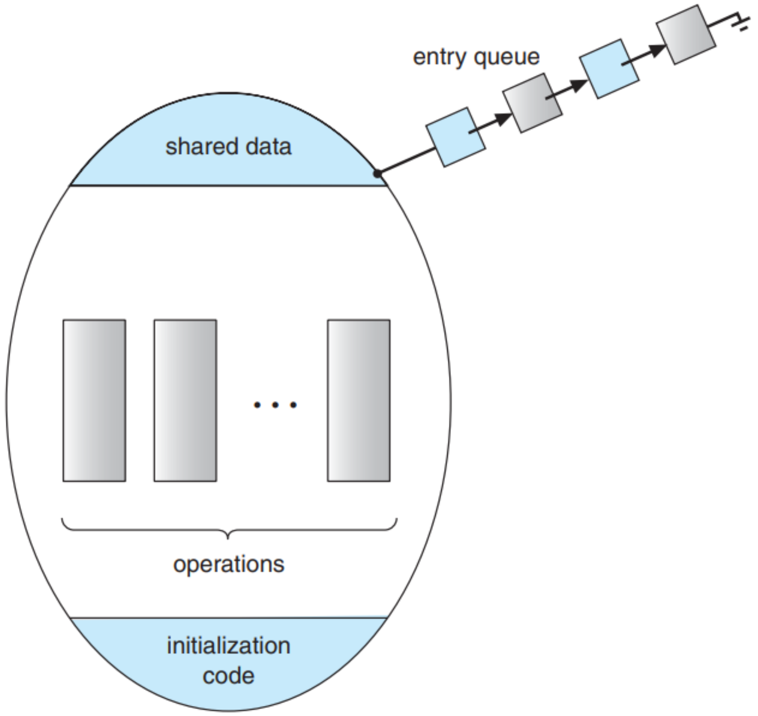
- Hoare 1974
- Monitor is an **abstract data type** designed for handling and defining shared resources in concurrent programming.
- A monitor is similar to a class that ties the data, operations, and in particular, the synchronization operations all together,
- Unlike classes,
  - monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor method at a time.
  - monitors require all data to be private.

Monitors: Components

- Shared Private Data
  - Represents the resource being managed.
  - This data cannot be accessed directly from outside the monitor, ensuring encapsulation and protection.

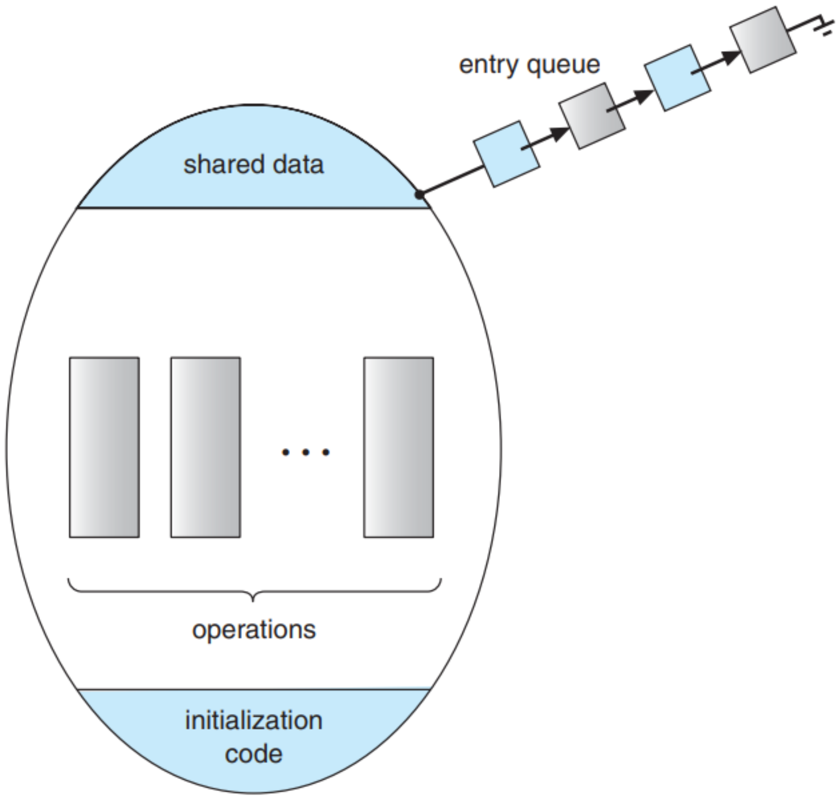
- Procedures that operate on the data
  - These are the gateway to accessing and manipulating the shared data.
  - Procedures can only operate on the data local to the monitor, promoting safe access patterns.
  - The monitors can have N procedures

- Synchronization primitives:
  - Monitors include mechanisms to synchronize access among threads that interact with the procedures.
  - This prevents race conditions and ensures mutual exclusion when accessing shared resources.



Monitor
Monitor Semantics

- Monitors guarantee mutual exclusion using locks.
  - Only one thread can execute a monitor procedure at any time.
    - “in the monitor”
  - If second thread invokes a monitor procedure at that time
    - It will block and wait for entry to the monitor
    - Need for a wait queue



Monitor
Condition Variables

- How can we change pop() to wait until something is on the queue?
  - Logically, we want to go to sleep inside of the critical section.
  - But if we hold on to the lock and sleep, then other threads cannot access the shared queue, add an item to it, and wake up the sleeping thread
  - => The thread could sleep forever

- **Solution**: use condition variables
  - Condition variables enable a thread to sleep inside a critical section/Monitor.
  - Any lock held by the thread is atomically released when the thread is put to sleep

```
Monitor monitor_name
{
      // shared variable declarations

      procedure P1(. . . .) {
          . . . .
      }

      procedure P2(. . . .) {
          . . . .
      }
      :
      procedure PN(. . . .) {
          . . . .
      }

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

For example:

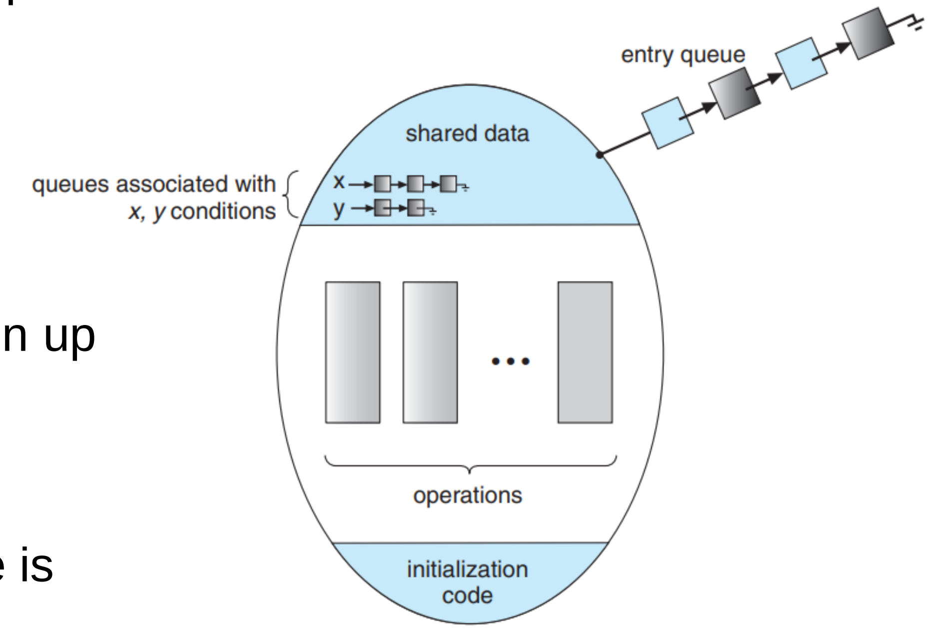
```
Monitor stack
{
      int top;
      void push(any_t *) {
          . . . .
      }

      any_t * pop() {
          . . . .
      }

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

Monitors
Synchronization Using Monitors

- Defines Condition Variables:
  - Condition variables are used to allow threads to wait for certain conditions to be met.
  - select a meaningful name (good programming practice)
    - condition x;

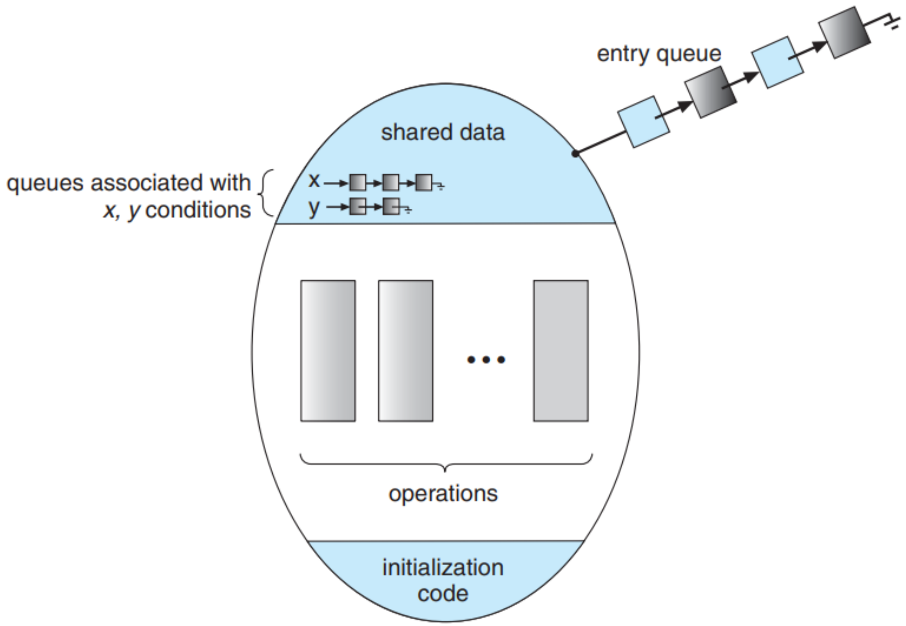
- 3 atomic operations on Condition Variables
  - x.wait(cond_var_name): release monitor lock, sleep until woken up
    - Condition variables have a waiting queue
  - x.notify/signal(cond_var_name): wake one process waiting on condition.
    - Notifier gives up lock and waiter runs immediately (if there is one)
  - x.notifyAll/signalall(cond_var_name): wake all processes waiting on condition
    - Useful for resource manager

- Rule: thread must hold the lock when doing condition variable operations.



Monitors
Types of wait queues

- Monitors have two kinds of “wait” queues
  - Entry to the monitor: has a queue of threads waiting to obtain mutual exclusion so they can enter
  - Condition variables: each condition variable has a queue of threads waiting on the associated condition



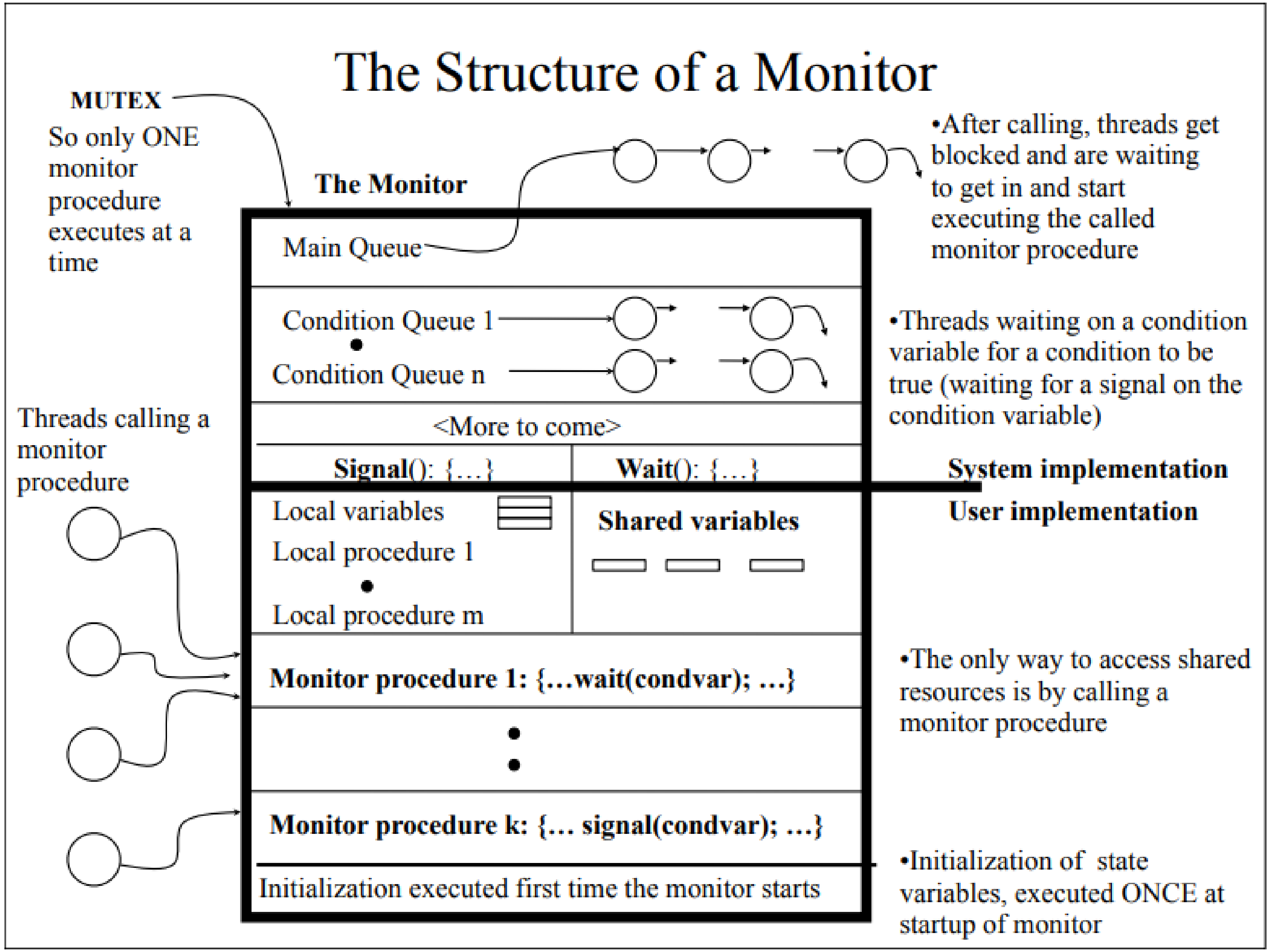
Monitor
Mesa versus Hoare Monitors

- What should happen when notify() is called?
  - If there is no waiting threads => the notifier continues and the signal is effectively lost (unlike what happens with semaphores).
  - If there is a waiting thread, one of the threads starts executing, others must wait

- **Mesa-style: (Nachos, Java, and most real operating systems(Linux, Windows, and macOS))**
  - The thread that notifies keeps the lock (and thus the processor).
  - The waiting thread waits for the lock.

- **Hoare-style: (most textbooks)**
  - The thread that notifies gives up the lock and the waiting thread gets the lock.
  - When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signaling thread.

Monitor
Summary: The Structure of a Monitor



Monitor limitations

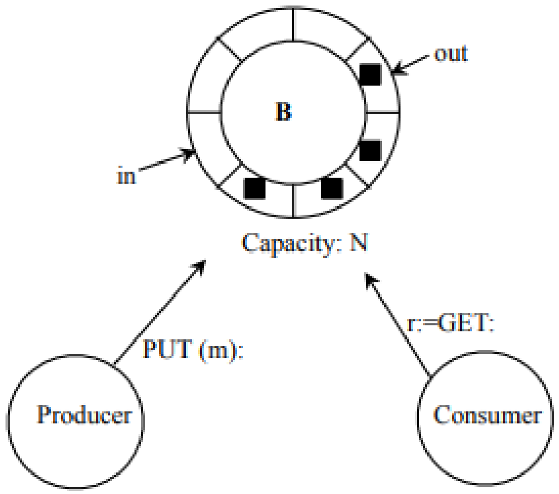
- Nested Monitors: Using one monitor within another can lead to deadlocks.
- Priority Inversion: Monitors do not address priority inversion, where a low-priority thread holding a monitor prevents a higher-priority thread from proceeding.
- Limited Flexibility: Monitors are less flexible than semaphores for certain complex synchronization needs.

Monitors: Implementation

- Monitors are a higher-level synchronization construct that requires deep integration with the language's concurrency model.
  - Languages with Monitor Support: Java, Python (with limitations), C#, and Ada.
  - Languages without Monitor Support: C, C++, JavaScript, Go, and others.

Monitors
Pseudocode Example: Monitor for producer consumer Bounded Buffer

Bounded Buffer Monitor



Rules for the buffer B:

- No Get when empty
- No Put when full
- B shared, so must have mutex between Put and Get

One condition variable for each condition:

- nonempty
- nonfull
- MUTEX is already provided by the monitor

```
/*Local  functions, variables*/
int in, out;
/*Shared variable*/
int B(0..n-1), count;
/*Condition variable*/
Condition nonfull, nonempty;

Put (int m):
{  if (count=n) wait (nonfull);
   B(in):=m;
   in:=in+1 MOD n;          /* MOD is % */
   count++;
   signal (nonempty);   }

int Get:
{  if (count=0) wait (nonempty);
   Get:=B(out);
   out:=out+1 MOD n;
   count--;
   signal (nonfull);  }

/* Initialization code*/
in:=out:=count:=0;
nonfull, nonempty:=EMPTY;
```

Monitors
Dining-Philosophers Solution Using Monitors

- Let's illustrate monitor concepts by presenting **a deadlock-free** solution to the dining philosophers problem.

- This solution imposes the restriction that a philosopher may pick up his chopsticks only if both of the are available.

- To code this solution,
  - we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:
    - `enum{thinking,hungry,eating} state[5];`
  - Philosopher i can set the variable `state[i]=eating` only if his two neighbors are not eating:
    - `(state [(i+4) % 5] !=eating (right) and (state [(i+1)%5!=eating (left));`

  - We also need to declare `condition self[5]` where philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he needs.

Monitors
Dining-Philosophers Solution Using Monitors

```
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;
    }
}
```

	P0	P1	P2	P3	P4
State[N]	T	T	T	T	T

```
int main()
{

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

return 0;
}
```

Fig: A monitor solution to the dining-philosopher problem

Monitors
Dining-Philosophers Solution Using Monitors

```
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;
    }
}
```

State[N]

P0	P1	P2	P3	P4
T	T	H	T	T

```
int main()
{

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

return 0;
}
```

Fig: A monitor solution to the dining-philosopher problem

Monitors
Dining-Philosophers Solution Using Monitors

```
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;
    }
}
```

	P0	P1	P2	P3	P4
State[N]	T	T	E	T	T

```
int main()
{

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

return 0;
}
```

Fig: A monitor solution to the dining-philosopher problem

Monitors
Dining-Philosophers Solution Using Monitors

```
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;
    }
}
```

	P0	P1	P2	P3	P4
State[N]	T	T	E	H	T

```
int main()
{

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

return 0;

```

If P3 try to eat it will be suspended

Fig: A monitor solution to the dining-philosopher problem

Monitors
Dining-Philosophers Solution Using Monitors

```
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;
    }
}
```

	P0	P1	P2	P3	P4
State[N]	T	T	T	E	T

```
int main()
{

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

return 0;
}
```

P3 weeks up when P2 finished eating.

Fig: A monitor solution to the dining-philosopher problem

Monitor
Semaphores versus Condition Variables

- Condition variables do not have any history, but semaphores do.
  - On a condition variable,
    - Signal, If no one is waiting, the signal is a no-op.
    - If a thread then does a condition Wait, it waits.
  - On a semaphore,
    - Signal, If no one is waiting, the value of the semaphore is incremented.
    - If a thread then does a semaphore Wait, then value is decremented of semaphore. If it is less than zero blocked else continues.

- Semaphore Wait and Signal are commutative, the result is the same regardless of the order of execution.

- Condition variables are not, and as a result they must be in a critical section to access shared variables and do their job.

- It is possible to implement monitors with semaphores

Quiz

Recap
Take-Home Message

- Modern Operating Systems (Tanenbaum): **Chapter 2(2.3.5, 2.5.1)**
- Operating System Concepts (Silberschatz ninth Edition):**Chapter 5(5.8)**
- Operating Systems: Internals and Design Principles (Starlings): **Chapter 5(5.3, 5.6)**