

Operating System Concepts

Che-Wei Chang

chewei@mail.cgu.edu.tw

Department of Computer Science and Information Engineering, Chang Gung University

Contents

1. Introduction
2. System Structures
3. Process Concept
4. Multithreaded Programming
5. Process Scheduling
6. Synchronization
7. Deadlocks
8. Memory-Management Strategies
9. Virtual-Memory Management
10. File System
11. Implementing File Systems
12. Secondary-Storage Systems

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University





Chapter 6. Synchronization

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



A Consumer-Producer Example

- Producer

```
while (1) {
    while (counter == BUFFER_SIZE)
        ;
    produce an item in nextp;
    buffer[in] = nextp;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

- Consumer:

```
while (1) {
    while (counter == 0)
        ;
    nextc = buffer[out];
    out = (out +1) % BUFFER_SIZE;
    counter--;
    consume an item in nextc;
}
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Race Condition

- One counter++ and one counter--

  r1 = counter r2 = counter
  r1 = r1 + 1 r2 = r2 - 1
  counter = r1 counter = r2

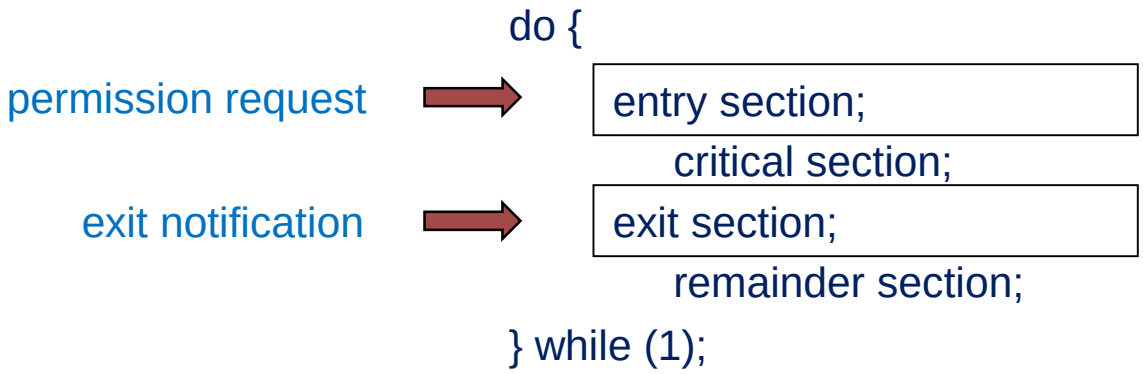
- Initially, let counter = 5
  1. P: r1 = counter
  2. P: r1 = r1 + 1
  3. C: r2 = counter
  4. C: r2 = r2 – 1 ⇒ A Race Condition!
  5. P: counter = r1
  6. C: counter = r2 = 4
- The result can be 4, 5 or 6

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Process Synchronization

- A Race Condition:
  - A situation where the outcome of the execution depends on the particular order of process scheduling
- The Critical-Section Problem:
  - Design a protocol that processes can use to cooperate
    - Each process has a segment of code, called a critical section, whose execution must be mutually exclusive
    - A general structure for the critical-section design



© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University

Solution of the Critical Section Problem

- Three Requirements
  - Mutual Exclusion:

    Only one process can be in its critical section
  - Progress:

    If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
  - Bounded Waiting:

    A waiting process only waits for a bounded number of processes to enter its critical section

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Peterson's Solution (1 / 5)

- Notation
  - Processes P_i and P_j
- Assumption
  - Every basic machine-language instruction is atomic
- Algorithm 1
  - Idea: Remember which process is allowed to enter its critical section. That is, P_i can enter its critical section if turn = i

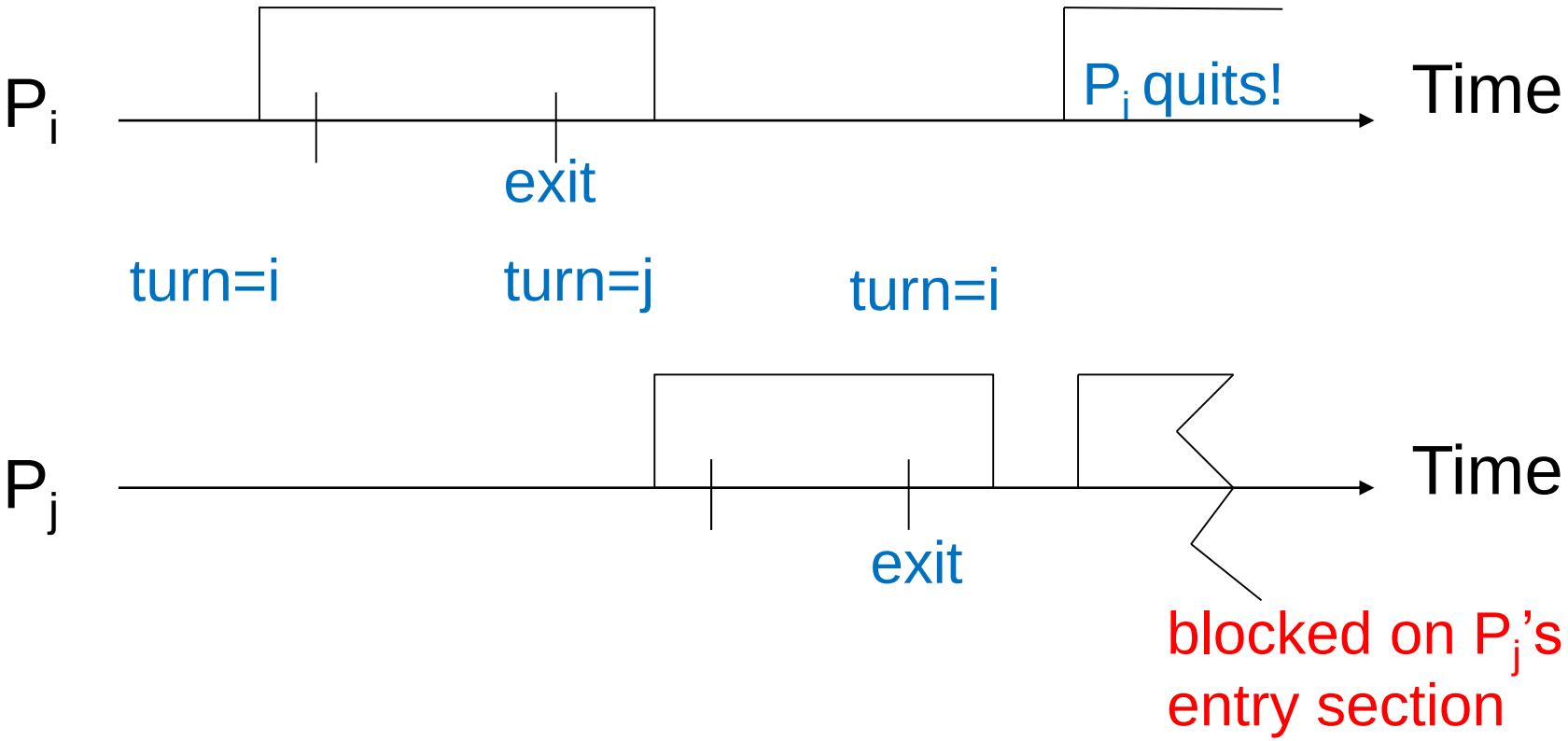
```
do {
    while (turn != i) ;
    critical section
    turn=j;
    remainder section
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Peterson's Solution (2/5)

- Algorithm 1 fails the progress requirement:



© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University

Peterson's Solution (3/5)

- Algorithm 2
  - Idea: Remember the state of each process
  - flag[i]==true → P_i is ready to enter its critical section
  - Algorithm 2 fails the progress requirement when flag[i] == flag[j] == true

Initially, flag[i]=flag[j]=false

```
do {
    flag[i]=true;
    while (flag[j]) ;
    critical section
    flag[i]=false;
    remainder section
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Peterson's Solution (4/5)

- Algorithm 3
  - Idea: Combine the ideas of Algorithms 1 and 2
  - When (flag[i] && turn=i), P_j must wait
  - Initially, flag[i]=flag[j]=false, and turn = i or j

```
do {
    flag[i]=true;
    turn=j;
    while (flag[j] && turn==j) ;
    critical section
    flag[i]=false;
    remainder section
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Peterson's Solution (5/5)

- Properties of Algorithm 3
  - Mutual Exclusion
    - The eventual value of *turn* determines which process enters the critical section
  - Progress
    - A process can only be stuck in the while loop, and the process which can keep it waiting must be in its critical sections
  - Bounded Waiting
    - Each process wait at most one entry by the other process

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Peterson's Solution (4/5)

Process P_i 的程式碼:

```
do {
    flag[i]=true;
    turn=j;
    while (flag[j] && turn==j) ;
    critical section
    flag[i]=false;
    remainder section
} while (1);
```

Process P_j 的程式碼:

```
do {
    flag[j]=true;
    turn=i;
    while (flag[i] && turn==i) ;
    critical section
    flag[j]=false;
    remainder section
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Brainstorming!

- Could we move turn=i; and turn=j; as follows:

Process P_i:

```
do {
    flag[i]=true;
    while (flag[j] && turn==j) ;
    critical section
    turn=j;
    flag[i]=false;
    remainder section
} while (1);
```

Process P_j:

```
do {
    flag[j]=true;
    while (flag[i] && turn==i) ;
    critical section
    turn=i;
    flag[j]=false;
    remainder section
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



答案是不行，卷哥Peterson應該也有想過這個問題。

- 不行的原因如下，如果照以下方式執行，Mutual Exclusion的條件會被違反：
  - 不失一般性，我們假設turn的初始值是i
  - P_j第一次開始執行時P_i尚未被執行過
    - 由於此時P_i還沒執行所以flag[i] 應為false
    - 所以P_j可以順利進入critical section
  - 在P_j進入critical section的這段期間內P_i也接著開始執行
    - 由於turn的初始值是i
    - 所以P_i也可以順利進入critical section
  - 這時候P_j和P_i同時在critical section裡 ➔違反Mutual Exclusion
  - 可是Peterson's Solution提出的做法(第一頁)不會有這個問題，大家可以自己想想

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Synchronization Hardware

- Motivation:
  - Hardware features make programming easier and improve system efficiency
- Approach:
  - Disable Interrupt → No Preemption
    - Infeasible in multiprocessor environments
    - Potential impacts on interrupt-driven system clocks
  - Atomic Hardware Instructions
    - Test-and-set, Swap, etc.

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Test and Set

Meaning

```
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target=true;
    return rv;
}
```

Usage

```
do {
    while (TestAndSet(&lock)) ;
        critical section
    lock=false;
        remainder section
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Swap

Meaning

```
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a=*b;
    *b=temp; }
```

Usage

```
do {
    key=true;
    while (key == true)
        Swap(&lock, &key);

        critical section

    lock=false;

        remainder section
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Case Study of Test and Set

- Problem
  - n tasks want to access some share data
- Mutual Exclusion
  - Pass if key== F or waiting[i]== F
- Progress
  - Exit process sends a process in
- Bounded Waiting
  - Wait at most n-1 times

```
do {
    waiting[i]=true;
    key=true;
    while (waiting[i] && key)
            key=TestAndSet(&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 (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Mutex Locks (1/2)

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Product critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Mutex Locks (2/2)

Meaning

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

Usage

```
do {
   acquire(lock);
      critical section
   release(lock);
      remainder section
} while (true);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Semaphores (1 / 3)

- Motivation:
  - A high-level solution for more complex problems
- Semaphore
  - A variable S only accessible by two atomic operations:

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

```
signal(S) {     /* V */
    S++;
}
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Semaphores (2/3)

- Critical Sections

```
do {
        wait(S);

        critical section

        signal(S);

        remainder section
} while (1);
```

- Precedence Enforcement

```
P1:
    S1;
    signal(S);

P2:
    wait(S);
    S2;
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Semaphores (3/3)

- Implementation
  - Spinlock: A Busy-Waiting Semaphore
    - "while (S <= 0)" causes the wasting of CPU cycles!
    - Advantage:
      - When locks are held for a short time, spinlocks are useful since no context switching is involved.
  - Semaphores with Blocked-Waiting
    - No busy waiting from the entry to the critical section!

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Semaphores with Block Waiting

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

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

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

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Deadlocks and Starvation

- Deadlock
  - A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set

```
P0:  wait(S);        P1: wait(Q);
     wait(Q);             wait(S);
       …                    …
     signal(S);           signal(Q);
     signal(Q);           signal(S);
```

- Starvation (or Indefinite Blocking)
  - e.g., a LIFO (last-in, first-out) queue

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Classical Problems of Synchronization

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



© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Bounded-Buffer Problem (1/2)

Producer:

```
do {
        produce an item in nextp;
        …….
        wait(empty);  /* control buffer availability */
        wait(mutex);  /* mutual exclusion */
        ……
        add nextp to buffer;
        signal(mutex);
        signal(full);  /* increase item counts */
} while (1);
```

empty is initialized to *n*

mutex is initialized to *1*

full is initialized to *0*

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Bounded-Buffer Problem (2/2)

Consumer:

```
do {
        wait(full);  /* control buffer availability */
        wait(mutex);  /* mutual exclusion */
        …….
        remove an item from buffer to nextp;
        ……
        signal(mutex);
        signal(empty);  /* increase item counts */
        consume nextp;
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Readers and Writers Problem (1/2)

- A data set is shared among a number of concurrent processes
  - Readers only read the data set; they do ***not*** perform any updates
  - Writers can both read and write
- Problem
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Readers and Writers Problem (2/2)

```
semaphore wrt, mutex;
  (initialized to 1);
int readcount=0;
```

Writer:

```
    wait(wrt);
    ……
    writing is performed
    ……
    signal(wrt)
```

Reader:

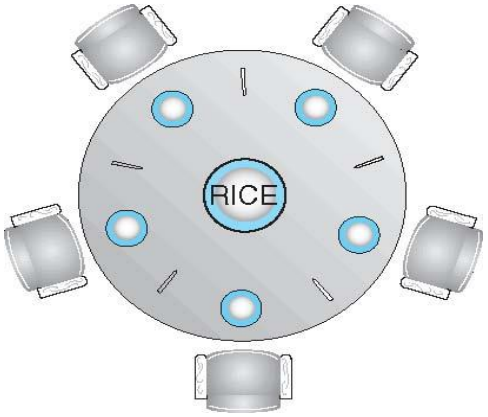
```
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
    … reading…
    wait(mutex);
    readcount--;
    if (readcount== 0)
        signal(wrt);
    signal(mutex);
```

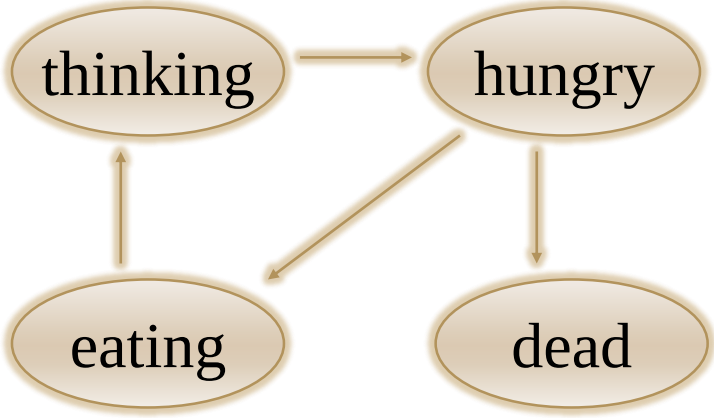
© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Dining-Philosopher Problem (1/3)

- Each philosopher must pick up one chopstick beside him/her at a time
- When two chopsticks are picked up, the philosopher can eat





© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University

Dining-Philosophers Problem (2/3)

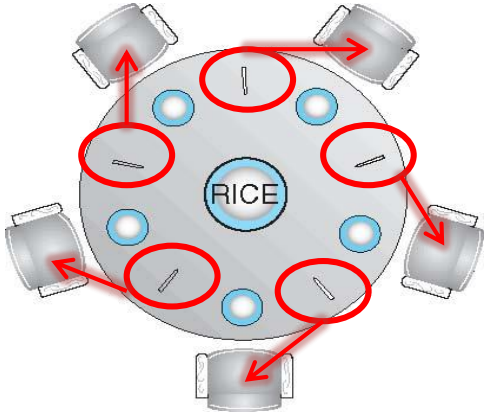
```
semaphore chopstick[5];
do {
        wait(chopstick[i]);
        wait(chopstick[(i + 1) % 5 ]);
        … eat …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        …think …
} while (1);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Dining-Philosophers Problem (3/3)

- This algorithm could create a deadlock
- Several possible remedies to the deadlock problem:
  - Allow at most four philosopher
  - Allow a philosopher to pick up chopsticks only if both are available
  - Asymmetric solution



© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

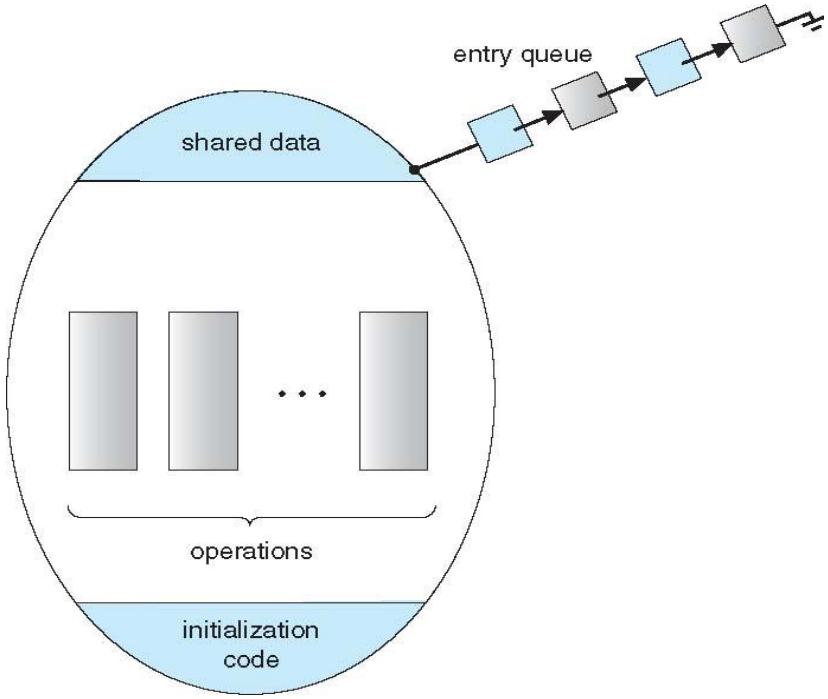
➔A high-level abstraction that provides a convenient and effective mechanism for process synchronization

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Monitor (1/2)

- Components
  - Variables
    - Monitor states
  - Procedures
    - Only access local variables or formal parameters

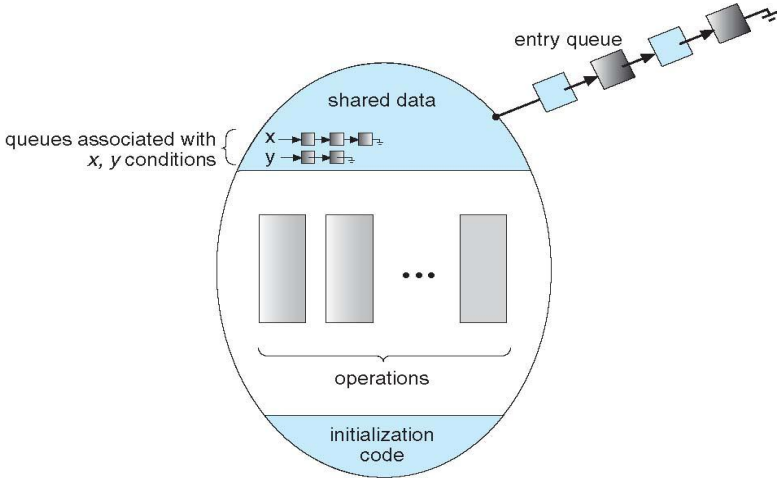



```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
   procedure Pn (…) {……}
   Initialization code (…) { … }
}
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University

Monitor (2/2)

- Condition Variables
  - x.wait () – a process that invokes the operation is suspended until x.signal ()
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - If no x.wait () on the variable, then it has no effect on the variable



© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University

Solution to Dining Philosophers (1/2)

```
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 left and right neighbors
           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;
    }
}
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Solution to Dining Philosophers (2/2)

- Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);

Eat

DiningPhilosophers.putdown (i);
```

- No deadlock, but starvation is possible

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Monitor Implementation Using Semaphores

- Semaphores
  - *mutex* – to protect the monitor
  - *next* – being initialized to zero, on which processes may suspend themselves
    - *next-count*
- For each external function *F*

```
wait(mutex);
...
body of F;
...
if (next-count > 0)
   signal(next);
else signal(mutex);
```

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Monitor Implementation Using Condition Variables

- For every condition *x*
  - A semaphore *x-sem*
  - An integer variable *x-count*
  - Implementation of x.wait() and x.signal :

- x.wait()

```
x-count++;
if (next-count > 0)
        signal(next);
else
        signal(mutex);
wait(x-sem);
x-count--;
```

- x.signal()

```
if (x-count > 0)
{
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

* x.wait() and x.signal() are invoked within a monitor

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University



Resuming Processes within a monitor

- How do we determine which of the suspended processes should be resumed next ?
  - FCFS ordering
  - Conditional-wait construct x.wait(c);
  - Monitor-scheduling algorithm
    - Built-in

      or
    - User define

© All Rights Reserved, Prof. Che-Wei Chang,
Department of Computer Science and Information Engineering, Chang Gung University

