Operating System: Chap6 Process Synchronization

National Tsing Hua University

2021, Fall Semester

Overview

- Background
- Critical Section
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Thread Programming
- Atomic Transactions

Background

- **Concurrent** access to shared data may result in **data inconsistency**
- Maintaining data consistency requires mechanism to ensure the orderly execution of cooperating processes

Consumer & Producer Problem

- Determine whether buffer is empty or full
  - Previously: use *in*, *out* position
  - Now: use count value

```
/*producer*/
while (1) {
    nextItem = getItem( );
    while (counter == BUFFER_SIZE) ;
    buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
/*consumer*/
while (1) {
    while (counter == 0) ;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Concurrent Operations on *counter*

- The statement "*counter++*" may be implemented in machine language as:

  move ax, counter

  add ax, 1

  move counter, ax

- The statement "*counter--*" may be implemented as:

  move bx, counter

  sub bx, 1

  move counter, bx

Instruction Interleaving

- Assume counter is initially 5. One interleaving of statement is:

  producer: move ax, counter → ax = 5
  producer: add ax, 1 → ax = 6
  context switch
  consumer: move bx, counter → bx = 5
  consumer: sub bx, 1 → bx = 4
  context switch
  producer: move counter, ax → counter = 6
  context switch
  consumer: move counter, bx → counter = 4

- The value of counter may be either 4, 5, or 6, where the correct result should be 5

Race Condition

- **Race condition**: the situation where several processes access and manipulate **shared data** concurrently. The final value of the shared data depends upon which process **finishes last**
- To prevent race condition, **concurrent** processes must be **synchronized**
  - On a single-processor machine, we could disable interrupt or use non-preemptive CPU scheduling
- Commonly described as **critical section problem**

Critical Section

The Critical-Section Problem

- *Purpose: a protocol for processes to cooperate*
- *Problem description:*
  - *N* processes are competing to use some shared data
  - Each process has a **code segment**, called ***critical section***, in which the shared data is accessed
  - Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section → ***mutually exclusive***

The Critical-Section Problem

- General code section structure
  - Only one process can be in a critical section

```
do {
    entry section          <- Get entry permission
        critical section   <- Modify shared data
    exit section           <- Release entry permission
        remainder section
} while (1);
```

Critical Section Requirements

1. **Mutual Exclusion**: if process P is executing in its CS, no other processes can be executing in their CS
2. **Progress**: if no process is executing in its CS and there exist some processes that wish to enter their CS, these processes cannot be postponed indefinitely
3. **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS

➔ How to design entry and exist section to satisfy the above requirement?

Review Slides (1)

- Race condition?
- Critical-Section (CS) problem? 4 sections?
  - entry, CS, exit, remainder
- 3 requirements for solutions to CS problems?
  - mutual exclusion
  - progress
  - bounded waiting

Critical Section Solutions & Synchronization Tools

- Software Solution
- Synchronization Hardware
- Semaphore
- Monitor

Algorithm for Two Processes

- Only 2 processes, P_0 and P_1
- Shared variables
  - int turn; //initially turn = 0
  - turn = $i \Rightarrow P_i$ can enter its critical section

```
/* Process 0 */
do {
  while (turn != 0) ;
    critical section
  turn = 1;
    remainder section
} while (1)
```

entry section

exit section

```
/* Process 1 */
do {
  while (turn != 1) ;
    critical section
  turn = 0;
    remainder section
} while (1)
```

➔ Mutual exclusion? Yes Progress? No
Bounded-Wait? Yes

Peterson's Solution for Two Processes

- Shared variables
  - int turn; //initially turn = 0
  - turn = $i \Rightarrow P_i$ can enter its critical section
  - boolean flag[2]; //initially flag [0] = flag [1] = false
  - flag [i] = true $\Rightarrow P_i$ ready to enter its critical section

```
//Pi:
do {
    flag[ i ] = TRUE;
    turn = j ;
    while (flag [ j ] &&
              turn == j ) ;
      critical section
    flag [ i ] = FALSE ;
      remainder section
} while (1) ;
```

Enter CS when either:

1. a process gets its turn
2. the other process is not ready

Proof of Peterson's Solution

- Mutual exclusion:
  - If P_0 CS ➔ flag[1] == false || turn == 0
  - If P_1 CS ➔ flag[0] == false || turn == 1
- Assume both processes in CS ➔ flag[0] == flag[1] == true

  ➔turn==0 for P_0 to enter, turn==1 for P_1 to enter
  - However, "turn" will be either 0 or 1 because its value will be set for both *processes*, but only one value will last
  - **Therefore, P_0,P_1 can't in CS at the same time!**

```
/* process 0 */
do {
  flag[ 0 ] = TRUE;
  turn = 1 ;
  while (flag [ 1 ] && turn == 1 ) ;
    critical section
  flag [ 0 ] = FALSE ;
    remainder section
} while (1) ;
```

```
/* process 1 */
do {
  flag[ 1 ] = TRUE;
  turn = 0 ;
  while (flag [ 0 ] && turn == 0 ) ;
    critical section
  flag [ 1 ] = FALSE ;
    remainder section
} while (1) ;
```

Proof of Peterson's Solution

- Progress (e.g., P_0 wishes to enter its CS):
  - (1) If P_1 *is not ready* ➔ flag[1] = false ➔ P_0 can enter
  - (2) If both are *ready* ➔ flag[0] == flag[1] == true
    - If trun ==0 then P_0 enters, otherwise P_1 enters
  - **Either cases, some waiting process can enter CS!**

```
/* process 0 */
do {
  flag[ 0 ] = TRUE;
  turn = 1 ;
  while (flag [ 1 ] && turn == 1 ) ;
    critical section
  flag [ 0 ] = FALSE ;
    remainder section
}  while (1) ;
```

(2)

(1)

```
/* process 1 */
do {
  flag[ 1 ] = TRUE;
  turn = 0 ;
  while (flag [ 0 ] && turn == 0 ) ;
    critical section
  flag [ 1 ] = FALSE ;
    remainder section
}  while (1) ;
```

Proof of Peterson's Solution

- Bounded waiting (e.g., P_0 wishes to enter its CS):
  - (1) Once P_1 exits CS → flag[1]==false → P_0 can enter
  - (2) If P_1 exits CS && reset flag[1]=true
    - → turn==0 (overwrite P_0 setting)→ P_0 can enter
  - ➢ **P_0 won't wait indefinitely!**

```
/* process 0 */
do {
  flag[ 0 ] = TRUE;
  turn = 1 ;
  while (flag [ 1 ] && turn == 1 ) ;
     critical section
  flag [ 0 ] = FALSE ;
     remainder section
}  while (1) ;
```

(2)

(1)

```
/* process 1 */
do {
  flag[ 1 ] = TRUE;
  turn = 0 ;
  while (flag [ 0 ] && turn == 0 ) ;
     critical section
  flag [ 1 ] = FALSE ;
     remainder section
}  while (1) ;
```

Producer/Consumer Problem

- Producer process

```
while (TRUE) {
    entry-section( );
    nextItem = getItem( );
    while (counter == BUFFER_SIZE);
    buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
    computing();
    exit-section( );
}
```

- Consumer process

```
while (TRUE) {
    entry-section( );
    while (counter == 0) ;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    computing();
    exit-section( );
}
```

➔ Incorrect: deadlock, if consumer enters the CS first.

Producer/Consumer Problem

- Producer process

```
while (TRUE) {
    nextItem = getItem( );
    while (counter == BUFFER_SIZE);
    buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    entry-section( );
    counter++;
    computing();
    exit-section( );
}
```

- Consumer process

```
while (TRUE) {
    while (counter == 0);
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    entry-section( );
    counter--;
    computing();
    exit-section( );
}
```

➔ Correct but poor performance

Producer/Consumer Problem

- Producer process

```
while (TRUE) {
    nextItem = getItem( );
    while (counter == BUFFER_SIZE);
    buffer[in] = nextItem;
    in = (in + 1) % BUFFER_SIZE;
    entry-section( );
    counter++;
    exit-section( );
    computing();
}
```

- Consumer process

```
while (TRUE) {
    while (counter == 0) ;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    entry-section( );
    counter--;
    exit-section( );
    computing();
}
```

➔ Correct & Maximize concurrent performance

Bakery Algorithm (*n* processes)

- Before enter its CS, each process receives a #
- Holder of the smallest # enters CS
- The numbering scheme always generates # in non-decreasing order; i.e., 1,2,3,3,4,5,5,5
- If processes P_i and P_j receive the same #, if $i < j$, then P_i is served first
- Notation:
  - $(a, b) < (c, d)$ if $a < c$ or if $a == c$ && $b < d$

Bakery Algorithm (*n* processes)

Get ticket

FCFS

release ticket

```
//Process i:
do {
  choosing [ i ] = TRUE ;
  num[ i ] = max(num[0],num[1],…,num[n-1]) + 1;
  choosing [ i ] = FALSE ;
  for (j = 0; j < n; j++) {
     while (choosing [ j ] ) ;
     while ((num[ j ] != 0) &&
               ((num[ j ], j) < (num[ i ], i))) ;
  }
     critical section
  num[ i ] = 0 ;
     reminder section
} while (1) ;
```

Cannot compare when num is being modified

- Bounded-waiting because processes enter CS on a First-Come, First Served basis

Bakery Algorithm (*n* processes)

- Why cannot compare when num is being modified?
- Without locking…
  - 1. Let 5 be the current maximum number
  - 2. If P1 and P4 take number together, but P4 finishes before P1
    - **P1 = 0**; P4 = 6 → P4 will enter the CS
  - 3. After P1 takes the number
    - P1 = P4 = 6 → P1 will enter the CS as well!!!
- With locking…
  - P4 will have to wait until P1 finish taking the number
  - Both P1 & P4 will have the new number “6” before comparison

Synchronization HW

Hardware Support

- The CS problem occurs because the modification of a shared variable may be interrupted
- If disable interrupts when in CS…
  - not feasible in multiprocessor machine
  - clock interrupts cannot fire in any machine
- HW support solution: **atomic instructions**
  - atomic: as one uninterruptible unit
  - examples: **TestAndSet**(var), **Swap**(a,b)

Atomic TestAndSet()

```
boolean TestAndSet ( bool  &lock) {
    bool  value = lock ;
    lock = TRUE ;
    return value ;
}
```

execute atomically: return the value of "lock" and set "lock" to TRUE

Mutual exclusion? Yes   Progress? Yes   Bounded-Wait? No!

Shared data: boolean lock; //initially lock = FALSE;

```
do {   // P0
    while (TestAndSet (lock) ) ;
        critical section
    lock = FALSE;
        remainder section
} while (1) ;
```

```
do {   // P1
    while (TestAndSet (lock) ) ;
        critical section
    lock = FALSE;
        remainder section
} while (1) ;
```

obtain lock

release lock

Atomic Swap()

- Idea: enter CS if lock==false:

Shared data: boolean **lock**; //initially **lock = FALSE**;

```
do {  // P0
    key0 = TRUE;
    while (key0 == TRUE)
        Swap (lock, key0) ;
      critical section
    lock = FALSE;
      remainder section
} while (1) ;
```

```
do {  // P1
    key1 = TRUE;
    while (key1 == TRUE)
        Swap (lock, key1) ;
      critical section
    lock = FALSE;
      remainder section
} while (1) ;
```

Mutual exclusion? Yes Progress? Yes Bounded-Wait? No!

Review Slide (2)

- Use software solution to solve CS?
  - Peterson’s and Bakery algorithms
- Use HW support to solve CS?
  - TestAndTest(), Swap()

Semaphores

Semaphore

- A tool to generalize the synchronization problem (easier to solve, but no guarantee for correctness)
- More specifically…
  - a record of how many units of a particular resource are available
    - If #record = 1 ➔ binary semaphore, mutex lock
    - If #record > 1 ➔ counting semaphore
  - accessed only through 2 *atomic* ops: **wait** & **signal**
- Spinlock implementation:
  - Semaphore is an integer variable

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

busy waiting

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

POSIX Semaphore

- Semaphore is part of POSIX standard BUT it is not belonged to Pthread
  - It can be used with or **without** thread
- POSIX Semaphore routines:
  - **sem_init**(sem_t *sem, int pshared, unsigned int value)
  - **sem_wait**(sem_t *sem)
  - **sem_post**(sem_t *sem)
  - **sem_getvalue**(sem_t *sem, int *valptr)
  - **sem_destory**(sem_t *sem)

Initial value of the semaphore

Current value of the semaphore

- Example:

```
#include <semaphore.h>
sem_t  sem;
sem_init(&sem);
sem_wait(&sem);
    // critical section
sem_post(&sem);
sem_destory(&sem);
```

n-Process Critical Section Problem

- shared data:

  semaphore mutex ; // initially mutex = 1

- Process P_i:

```
do {
    wait (mutex) ;  // pthread_mutex_lock(&mutex)
        critical section
    signal (mutex); // pthread_mutex_unlock(&mutex)
        remainder section
} while (1) ;
```

Progress? Yes

Bounded waiting? Depends on the implementation of wait()

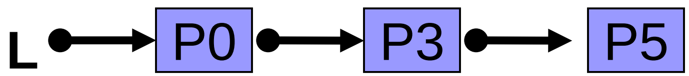
Non-busy waiting Implementation

- Semaphore is data struct with a queue
  - may use any queuing strategy (FIFO, FILO, etc)

```
typedef struct {
    int value; // init to 0
    struct process *L ;
    // "PCB" queue
} semaphore ;
```

E.g.,:

Value = -3



- wait() and signal()
  - use system calls: block() and wakeup()
  - must be executed **atomically**

```
void wait (semaphore S) {
  S.value--; // subtract first
  if (S.value < 0) {
    add this process to S.L ;
    block( );
  }
}
```

```
void signal (semaphore S) {
  S.value++;
  if (S.value <= 0) {
    remove a process P from S.L ;
    wakeup(P);
  }
}
```

Atomic Operation

- How to ensure atomic wait & signal ops?
  - Single-processor: disable interrupts
  - Multi-processor:
    - HW support (e.g. Test-And-Set, Swap)
    - SW solution (Peterson's solution, Bakery algorithm)

Semaphore with Critical Section

```
void wait (semaphore S) {
    entry-section( );
    S.value--;
    if (S.value < 0) {
      add this process to S.L ;
      exit-section( );
      sleep( );
   }
   else {
      exit-section( );
   }
}
```

```
void signal (semaphore S) {
   entry-section( );
   S.value++;
   if (S.value <= 0)
      remove a process P from S.L;
      exit-section( );
      wakeup(P);
   }
   else {
      exit-section( );
   }
}
```

- Busy waiting for entry-section()?
  - limited to only the CS of wait & signal (~10 instructions) ➔ very short period of time

Cooperation Synchronization

- P1 executes S1 ; P2 executes S2
  - S2 be executed only after S1 has completed
- Implementation:
  - shared var:

    semaphore sync ; // initially sync = 0

P1:	P2:
S1 ;	wait (sync) ;
signal (sync) ;	S2 ;

A More Complicated Example

(Initially, all semaphores are 0)

begin

P_1: S_1; signal(a); signal(b);

P_2: wait(a); S_2; signal(c);

P_3: wait(b); S_3; signal(d);

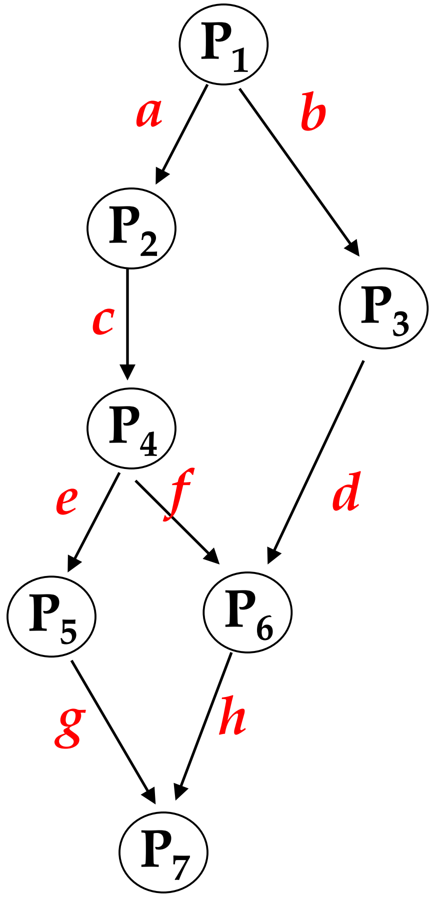
P_4: wait(c); S_4; signal(e); signal(f);

P_5: wait(e); S_5; signal(g);

P_6: wait(f); wait(d); S_6; signal(h);

P_7: wait(g); wait(h); S_7;

end



Deadlocks & Starvation

- Deadlocks: 2 processes are waiting indefinitely for each other to release resources
- Starvation: example: LIFO queue in semaphore process queue

P_0	P_1
wait(S);	*wait(Q);*
wait(Q);	*wait(S);*
⋮	⋮
signal(S);	*signal(Q);*
signal(Q)	*signal(S);*

Review Slide (3)

- What’s semaphore? 2 operations?
- What’s busy-waiting (spinlock) semaphore?
- What’s non-busy-waiting (non-spinlock) semaphore?
- How to ensure atomic wait & signal ops?
- Deadlock? starvation?

Classical Synchronization Problems

Listing & Purpose

- Purpose: used for testing newly proposed synchronization scheme
- Bounded-Buffer (Producer-Consumer) Problem
- Reader-Writers Problem
- Dining-Philosopher Problem

Bounded-Buffer Problem

- A pool of *n* buffers, each capable of holding one item
- Producer:
  - grab an empty buffer
  - place an item into the buffer
  - waits if no empty buffer is available
- Consumer:
  - grab a buffer and retracts the item
  - place the buffer back to the free pool
  - waits if all buffers are empty

Readers-Writers Problem

- A set of shared data objects
- A group of processes
  - reader processes (read shared objects)
  - writer processes (update shared objects)
  - **a writer process has exclusive access to a shared object**
- *Different variations involving priority*
  - *first RW problem*: no reader will be kept waiting unless a writer is updating a shared object
  - *second RW problem*: once a writer is ready, it performs the updates as soon as the shared object is released

    → writer has higher priority than reader

    → once a writer is ready, no new reader may start reading

First Reader-Writer Algorithm

```
// mutual exclusion for write
semaphore wrt=1
// mutual exclusion for readcount
semaphore mutex=1
int readcount=0;

Writer(){
    while(TRUE){
        wait(wrt);

          // Writer Code

        signal(wrt);
    }
}
```

```
Reader(){
  while(TRUE){
    wait(mutex);
      readcount++;
      if(readcount==1)
        wait(wrt);
    signal(mutex);

          // Reader Code

    wait(mutex);
      readcount--;
      if(readcount==0)
        signal(wrt);
    signal(mutex);
  }
}
```

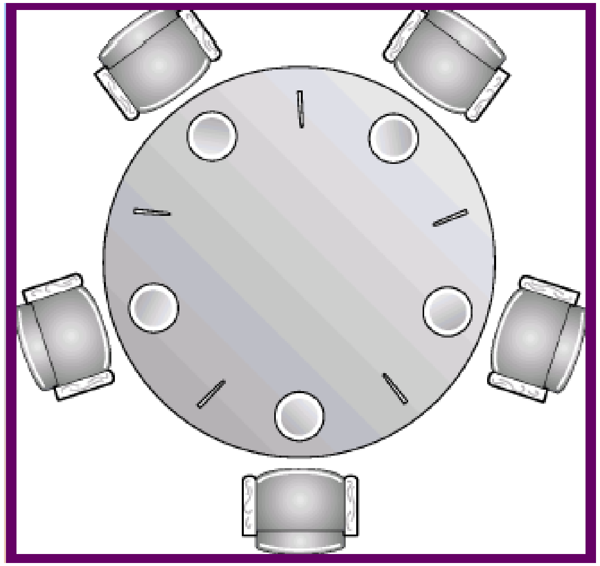
Acquire write lock if reads haven't → if(readcount==1) wait(wrt);

release write lock if no more reads → if(readcount==0) signal(wrt);

- Readers share a single wrt lock
- Writer may have starvation problem

Dining-Philosophers Problem

- 5 persons sitting on 5 chairs with 5 chopsticks
- A person is either thinking or eating
  - thinking: no interaction with the rest 4 persons
  - eating: need 2 chopsticks at hand
  - a person picks up 1 chopstick at a time
  - done eating: put down both chopsticks
- **deadlock problem**
  - one chopstick as one semaphore
- **starvation problem**



Monitors

Motivation

- Although semaphores provide a convenient and effective synchronization mechanism, its correctness is depending on the programmer
  - All processes access a shared data object must execute wait() and signal() in the right order and right place
  - This may not be true because honest programming error or uncooperative programmer

Monitor --- A high-level language construct

- The representation of a **monitor type** consists of
  - declarations of **variables** whose values define the state of an instance of the type
  - **Procedures/functions** that implement operations on the type
- The monitor type is similar to a class in O.O. language
  - A procedure within a monitor can access only local variables and the formal parameters
  - The local variables of a monitor can be used only by the local procedures
- But, the monitor ensures that only one process at a time can be **active** within the monitor
- Similar idea is incorporated to many prog. language:
  - concurrent pascal, C# and Java

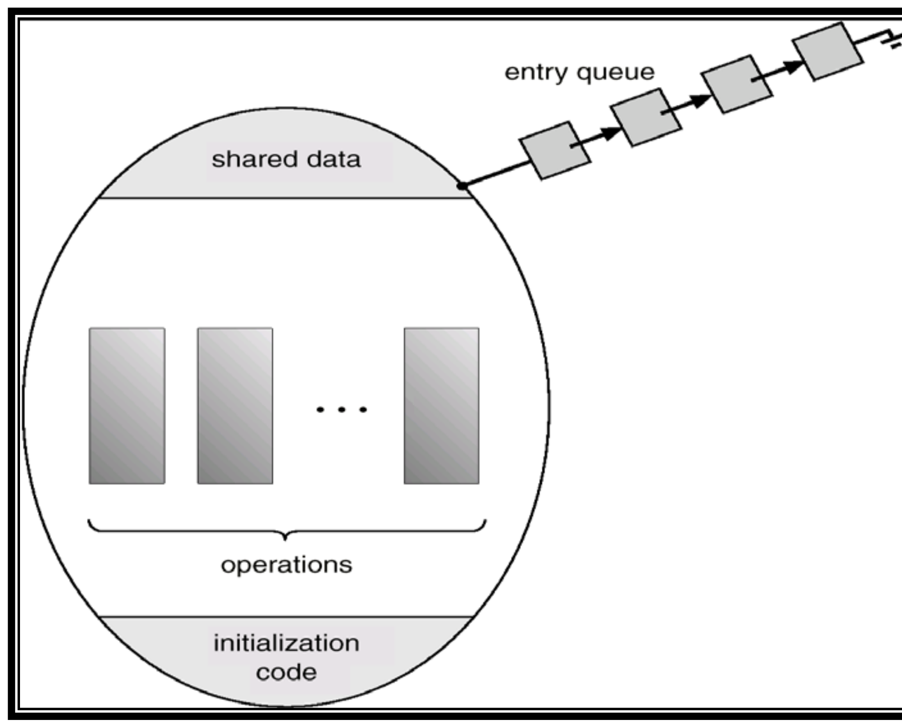
Monitor

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes

Syntax

```
monitor monitor-name {
        // shared variable declarations
        procedure body P1 (…) {
                . . .
        }
        procedure body P2 (…) {
                . . .
        }
        procedure body Pn (…) {
                . . .
        }
        initialization code {
        }
}
```

Schematic View



Monitor Condition Variables

- To allow a process to wait **within** the monitor, a condition variable must be declared, as

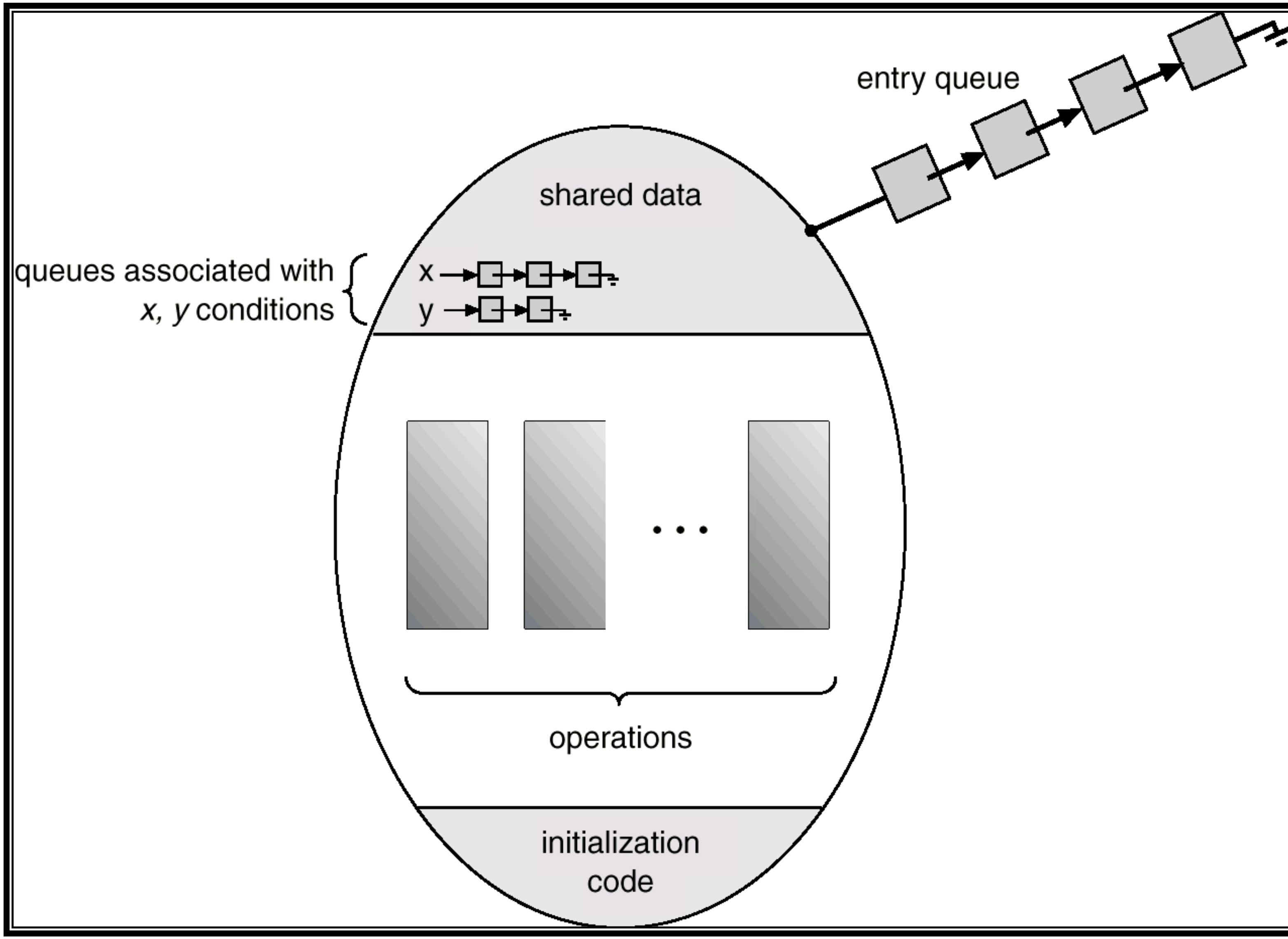
  condition x, y;

- Condition variable can only be used with the operations **wait()** and **signal()**
  - **x.wait();**
    means that the process invoking this operation is suspended until another process invokes
  - **x.signal();**

    resumes exactly one suspended process. If no process is suspended, then the signal operation **has no effect**

    (In contrast, signal always change the state of a semaphore)

Monitor With Condition Variables



Dining Philosophers Example

```
monitor dp {
  enum {thinking, hungry, eating} state[5]; //current state
  condition self[5]; //delay eating if can't obtain chopsticks
  void pickup(int i)          // pickup chopsticks
  void putdown(int i)         // putdown chopsticks
  void test(int i)            // try to eat
  void init() {
     for (int i = 0; i < 5; i++)
           state[i] = thinking;
  }
}
```

```
void pickup(int i) {
    state[i] = hungry;
    test(i); //try to eat
    if (state[i] != eating)
        self[i].wait();//wait to eat
}
```

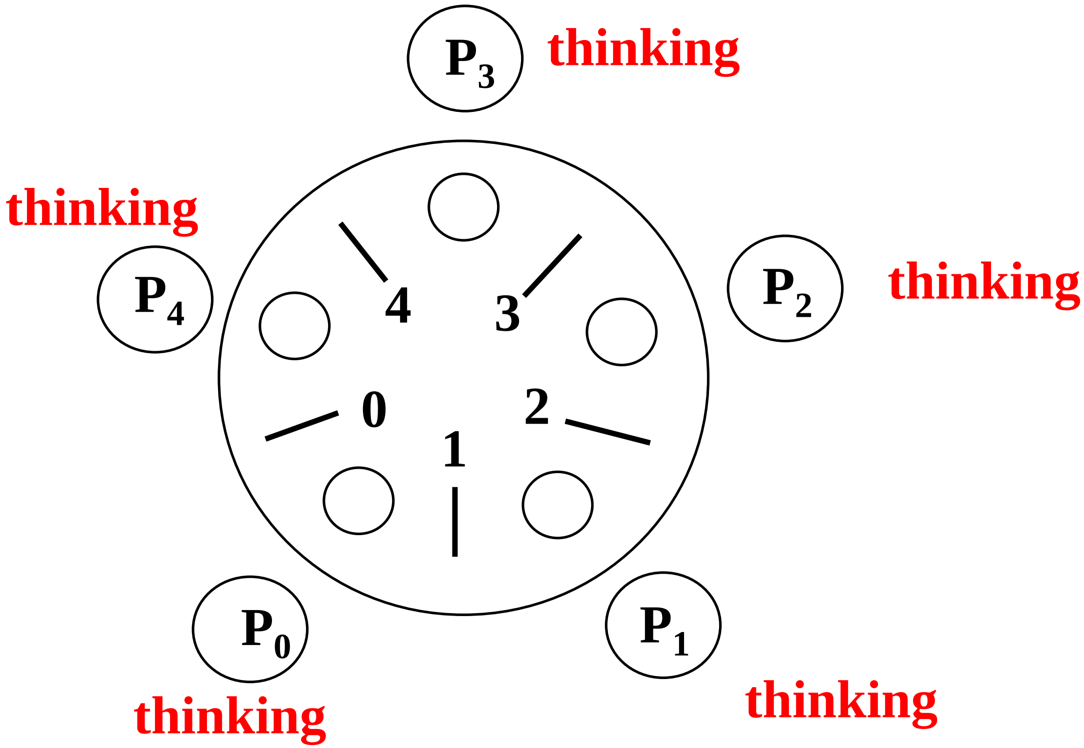
```
void putdown(int i) {
    state[i] = thinking;
    // check if neighbors
    // are waiting to eat
    test((i+4) % 5);
    test((i+1) % 5);
}
```

```
//try to let Pi eat (if it is hungry)
void test(int i) {
  if ( (state[(i + 4) % 5] != eating) &&(state[(i + 1) % 5] != eating)
     && (state[i] == hungry) ) {
       //No neighbors are eating and Pi is hungry
       state[i] = eating;
       self[i].signal();
  }
}
```

If P_i is suspended, resume it

If P_i is not suspended, **no effect**

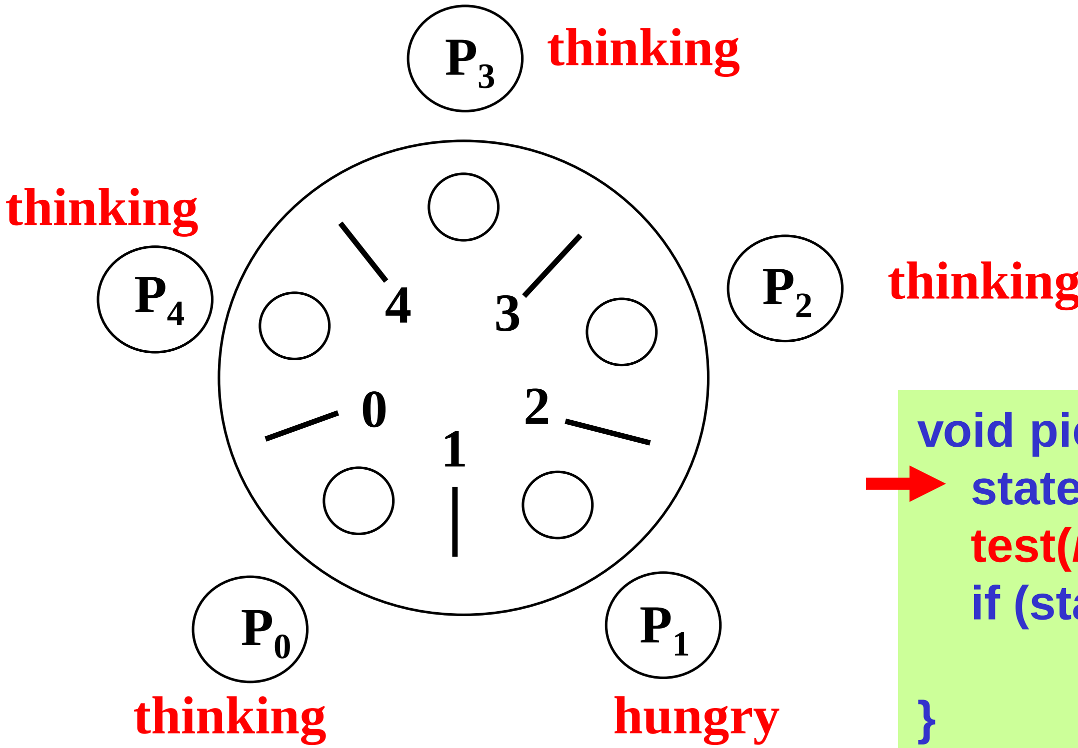
An illustration



P1:
DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration

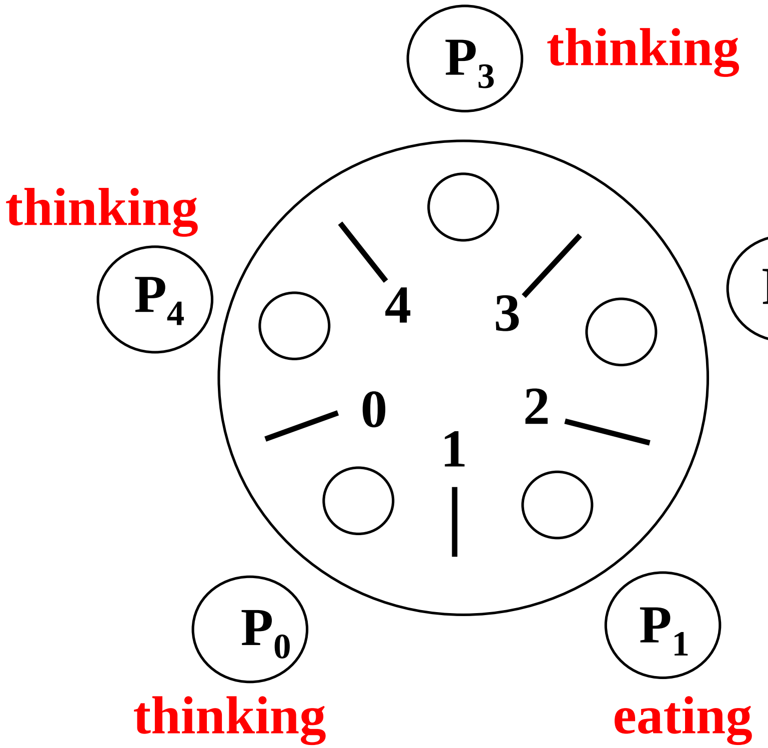



```
void pickup(int i) {
    state[i] = hungry;
    test(i); //try to eat
    if (state[i] != eating)
            self[i].wait();//wait to eat
}
```

P1:
DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration



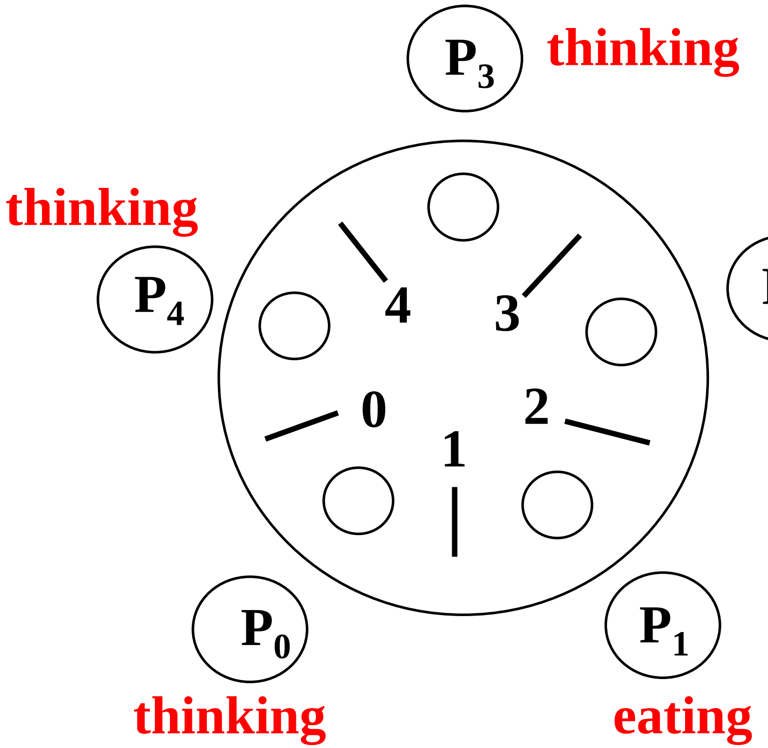
P1

```
void pickup(int i) {
    state[i] = hungry;
    test(i); //try to eat
    if (state[i] != eating)
            self[i].wait();//wait to eat
}
```

P1:
DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration



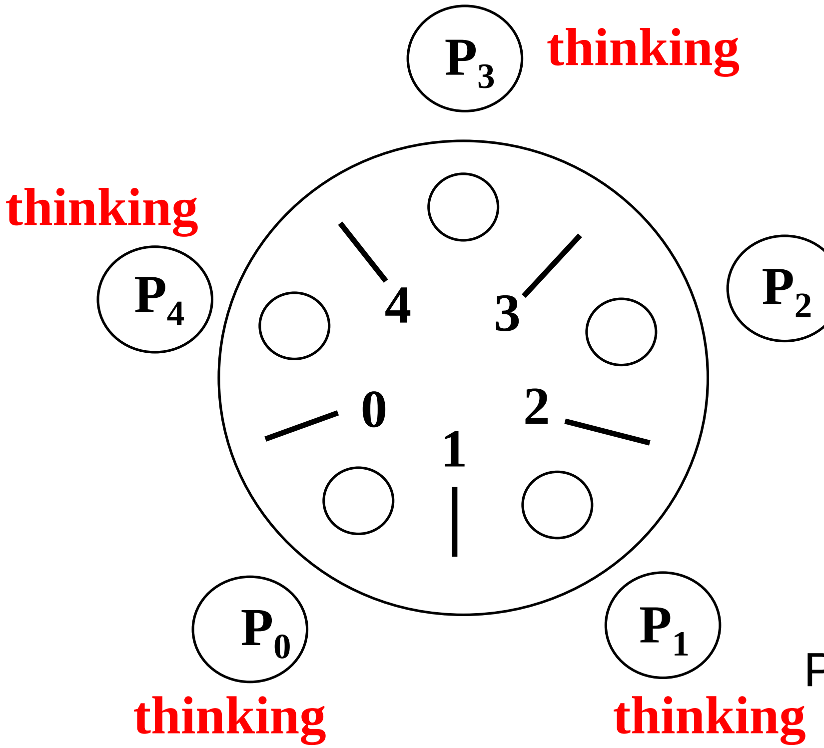
hungry ➔ self[2].wait

```
void pickup(int i) {
    state[i] = hungry;
    test(i); //try to eat
    if (state[i] != eating)
P2 →        self[i].wait();//wait to eat
}
```

P1:
DiningPhilosophers.pickup(1)
→ eat
DiningPhilosophers.putdown(1)

P2:
→ DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration



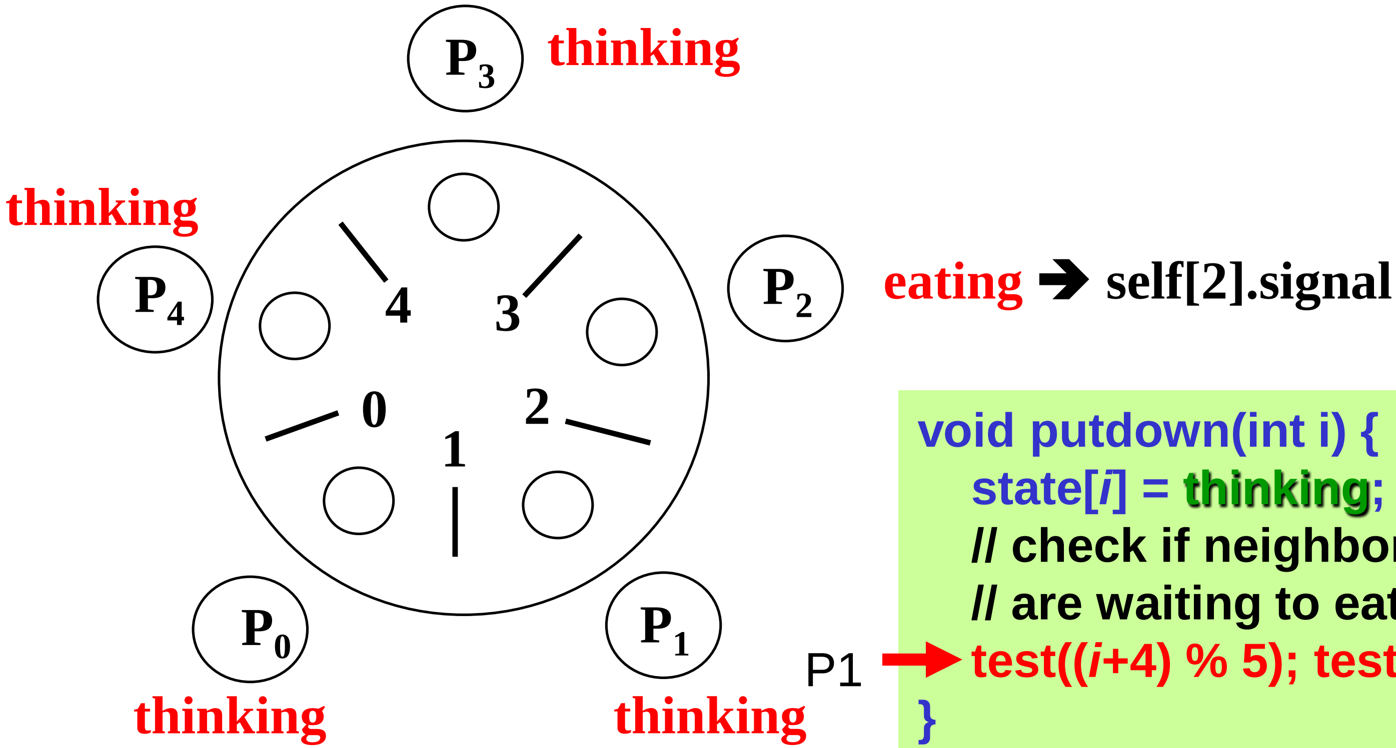
hungry ➔ self[2].wait

```
void putdown(int i) {
    state[i] = thinking;
    // check if neighbors
    // are waiting to eat
P1 ➔ test((i+4) % 5); test((i+1) % 5);
}
```

P1:
DiningPhilosophers.pickup(1)
eat
➔ DiningPhilosophers.putdown(1)

P2:
➔ DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration

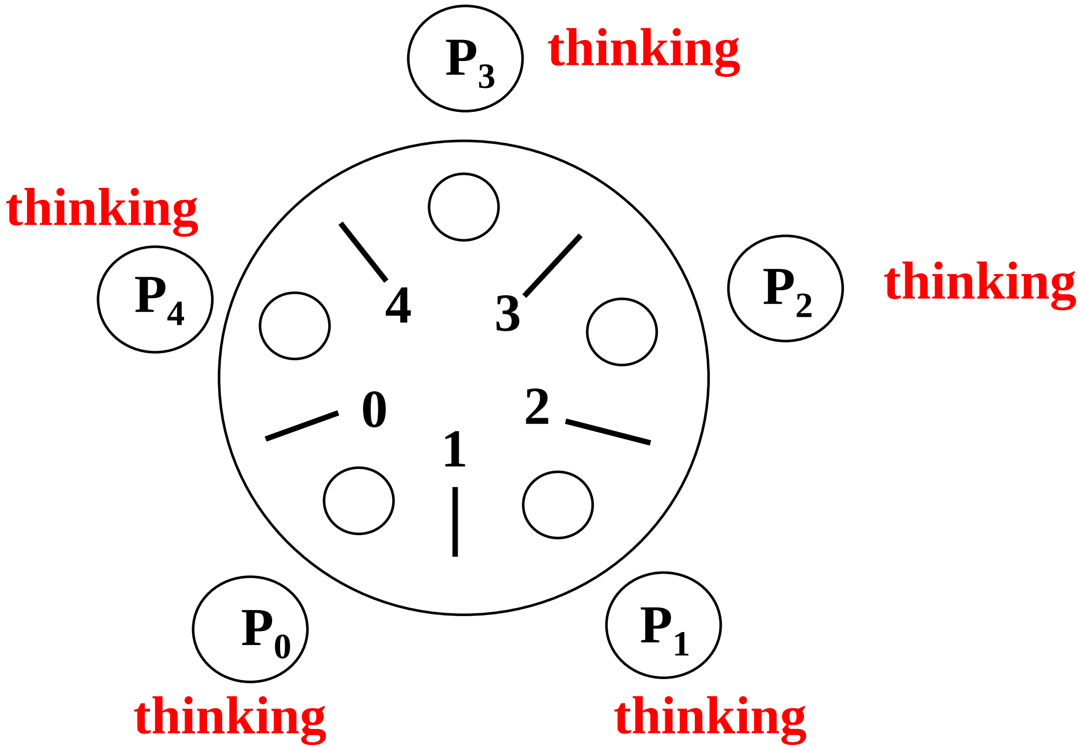


P1 ➔
```
void putdown(int i) {
    state[i] = thinking;
    // check if neighbors
    // are waiting to eat
    test((i+4) % 5); test((i+1) % 5);
}
```

P1:
DiningPhilosophers.pickup(1)
eat
➔ DiningPhilosophers.putdown(1)

P2:
➔ DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

An illustration



P1:
DiningPhilosophers.pickup(1)
eat
DiningPhilosophers.putdown(1)

P2:
DiningPhilosophers.pickup(2)
eat
DiningPhilosophers.putdown(2)

Thread Programming

Pthread Lock/Mutex Routines

- To use mutex, it must be declared as of type **pthread_mutex_t** and initialized with **pthread_mutex_init()**
- A mutex is destroyed with **pthread_mutex_destory()**
- A critical section can then be protected using **pthread_mutex_lock()** and **pthread_mutex_unlock()**
- Example:

```
#include "pthread.h"
pthread_mutex   mutex;
pthread_mutex_init (&mutex, NULL);       specify default attribute for the mutex
pthread_mutex_lock(&mutex);              // enter critical section

        Critical Section

pthread_mutex_unlock(&mutex);            // leave critical section
pthread_mutex_destory(&mutex);
```

Condition Variables (CV)

- CV represent some condition that a thread can:
  - Wait on, until the condition occurs; or
  - Notify other waiting threads that the condition has occurred
- Three operations on condition variables:
  - wait() --- Block until another thread calls signal() or broadcast() on the CV
  - signal() --- Wake up one thread waiting on the CV
  - broadcast() --- Wake up all threads waiting on the CV
- In Pthread, CV type is a pthread_cond_t
  - Use **pthread_cond_init()** to initialize
  - **pthread_cond_wait (&theCV, &somelock)**
  - **pthread_cond_signal (&theCV)**
  - **pthread_cond_broadcast (&theCV)**

Using Condition Variable

- Example:
  - A threads is designed to take action when x=0
  - Another thread is responsible for decrementing the counter

```
pthread_cond_t    cond;                  pthread_mutex_t    mutex;
pthread_cond_init (cond, NULL);          pthread_mutex_init (mutex, NULL);
```

```
action() {
   pthread_mutex_lock (&mutex)
   if (x != 0)
      pthread_cond_wait (cond, mutex);
   pthread_mutex_unlock (&mutex);
   take_action();
}
```

```
counter() {
  pthread_mutex_lock (&mutex)
  x--;
  if (x==0)
     pthread_cond_signal (cond);
  pthread_mutex_unlock (&mutex);
}
```

- All condition variable operation **MUST** be performed while a mutex is **locked**!!!

Using Condition Variable

```
action() {
  pthread_mutex_lock (&mutex)
  whild (x != 0)
    pthread_cond_wait (cond, mutex);
  pthread_mutex_unlock (&mutex);
  take_action();
}
```

```
counter() {
  pthread_mutex_lock (&mutex)
  x--;
  if (x==0)
    pthread_cond_signal (cond);
  pthread_mutex_unlock (&mutex);
}
```

- What really happens…

1. Lock mutex

Using Condition Variable

```
action() {
  pthread_mutex_lock (&mutex)
  whild (x != 0)
    pthread_cond_wait (cond, mutex);
  pthread_mutex_unlock (&mutex);
  take_action();
}
```

```
counter() {
  pthread_mutex_lock (&mutex)
  x--;
  if (x==0)
    pthread_cond_signal (cond);
  pthread_mutex_unlock (&mutex);
}
```

- What really happens…

1. Lock mutex
2. Wait()
    1. Put the thread into sleep & **releases the lock**

1. Lock mutex

Using Condition Variable

```
action() {
   pthread_mutex_lock (&mutex)
   whild (x != 0)
      pthread_cond_wait (cond, mutex);
   pthread_mutex_unlock (&mutex);
   take_action();
}
```

```
counter() {
  pthread_mutex_lock (&mutex)
  x--;
  if (x==0)
     pthread_cond_signal (cond);
  pthread_mutex_unlock (&mutex);
}
```

- What really happens…

1. Lock mutex
2. Wait()
    1. Put the thread into sleep & **releases the lock**
    1. Waked up, but the **thread is locked**

1. Lock mutex
2. Signal()

Using Condition Variable

```
action() {
   pthread_mutex_lock (&mutex)
   whild (x != 0)
      pthread_cond_wait (cond, mutex);
   pthread_mutex_unlock (&mutex);
   take_action();
}
```

```
counter() {
  pthread_mutex_lock (&mutex)
  x--;
  if (x==0)
     pthread_cond_signal (cond);
  pthread_mutex_unlock (&mutex);
}
```

- What really happens…

1. Lock mutex
2. Wait()
   1. Put the thread into sleep & **releases the lock**
   1. Waked up, but the **thread is locked**
   2. **Re-acquire lock** and **resume execution**

1. Lock mutex
2. Signal()
3. Releases the lock

Using Condition Variable

```
action() {
   pthread_mutex_lock (&mutex)
   whild (x != 0)
      pthread_cond_wait (cond, mutex);
   pthread_mutex_unlock (&mutex);
   take_action();
}
```

```
counter() {
  pthread_mutex_lock (&mutex)
  x--;
  if (x==0)
     pthread_cond_signal (cond);
  pthread_mutex_unlock (&mutex);
}
```

- What really happens…

1. Lock mutex
2. Wait()
    1. Put the thread into sleep & **releases the lock**
    1. Waked up, but the **thread is locked**
    2. **Re-acquire lock** and **resume execution**
3. Release the lock

1. Lock mutex
2. Signal()
3. Releases the lock

Using Condition Variable

```
action() {
   pthread_mutex_lock (&mutex)
   whild (x != 0)
      pthread_cond_wait (cond, mutex);
   pthread_mutex_unlock (&mutex);
   take_action();
}
```

```
counter() {
  pthread_mutex_lock (&mutex)
  x--;
  if (x==0)
     pthread_cond_signal (cond);
  pthread_mutex_unlock (&mutex);
}
```

- What really happens…

1. Lock mutex
2. Wait()
   1. Put the thread into sleep & **releases the lock**
   1. Waked up, but the **thread is locked**
   2. **Re-acquire lock** and resume execution
3. Release the lock

1. Lock mutex
2. Signal()
3. Releases the lock

Another reason why condition variable op. MUST within mutex lock

ThreadPool Implementation

Threadpool structure

```
struct threadpool_t {
    pthread_mutex_t lock;
    pthread_cond_t notify;
    pthread_t *threads;
    threadpool_task_t *queue;
    int thread_count;
    int queue_size;
    int head;
    int tail;
    int count;
    int shutdown;
    int started;
};
```

Task structure

```
typedef struct {
    void (*function)(void *);
    void *argument;
} threadpool_task_t;
```

Allocate thread and task queue

```
/* Allocate thread and task queue */
pool->threads = (pthread_t *) malloc(sizeof(pthread_t) * thread_count);
pool->queue = (threadpool_task_t *) malloc(sizeof(threadpool_task_t) * queue_size);
```

ThreadPool Implementation

```c
static void *threadpool_thread(void *threadpool)
 {
     threadpool_t *pool = (threadpool_t *)threadpool;
     threadpool_task_t task;

     for(;;) {
         /* Lock must be taken to wait on conditional variable */
         pthread_mutex_lock(&(pool->lock));

         /* Wait on condition variable, check for spurious wakeups.
            When returning from pthread_cond_wait(), we own the lock. */
         while((pool->count == 0) && (!pool->shutdown)) {
             pthread_cond_wait(&(pool->notify), &(pool->lock));
         }
```

thread handler function

ThreadPool Implementation

```
    /* Grab our task */
    task.function = pool->queue[pool->head].function;
    task.argument = pool->queue[pool->head].argument;
    pool->head += 1;
    pool->head = (pool->head == pool->queue_size) ? 0 : pool->head;
    pool->count -= 1;

    /* Unlock */
    pthread_mutex_unlock(&(pool->lock));

    /* Get to work */
    (*(task.function))(task.argument);
}
```

Synchronized Tools in JAVA

- Synchronized Methods (Monitor)
  - Synchronized method uses the method receiver as a lock
  - Two invocations of synchronized **methods cannot interleave on the same object**
  - When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block until the first thread exits the object

```
public class SynchronizedCounter {
        private int c = 0;
        public synchronized void increment() { c++; }
        public synchronized void decrement() { c--; }
        public synchronized int value() { return c; }
}
```

Synchronized Tools in JAVA

- Synchronized Statement (Mutex Lock)
  - Synchronized blocks uses the **expression** as a lock
  - A synchronized Statement can only be executed once the thread has obtained a lock for the object or the class that has been referred to in the statement
  - useful for improving concurrency **with fine-grained**

```
public void run()
{
    synchronized(p1)
    {
        int i = 10; // statement without locking requirement
        p1.display(s1);
    }
}
```

Review Slides (4)

- Bounded-buffer problem?
- Reader-Writer problem?
- Dining Philosopher problem?
- What is monitor and why need monitor?

Atomic Transactions

System Model

- **Transaction**: a collection of instructions (or instructions) that performs a single logic function
- **Atomic Transaction**: operations happen as a single logical unit of work, in its entirely, or not at all
- Atomic transaction is particular a concern for database system
  - Strong interest to use DB techniques in OS

File I/O Example

- Transaction is a series of read and write operations
- Terminated by **commit** (transaction successful) or **abort** (transaction failed) operation
- Aborted transaction must be **rolled back** to undo any changes it performed
  - It is part of the responsibility of the system to ensure this property

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
  - **Stable storage**: never lost its stored data
- **Write-ahead logging:** Each log record describes single transaction write operation
  - Transaction name
  - Data item name
  - Old & new values
  - Special events: <T_i starts>, <T_i commits>
- Log is used to reconstruct the state of the data items modified by the transactions
  - Use undo (T_i), redo(T_i) to recover data

Checkpoints

- When failure occurs, must consult the log to determine which transactions must be re-done
  - Searching process is time consuming
  - Redone may not be necessary for all transactions
- Use **checkpoints** to reduce the above overhead:
  - Output all log records to stable storage
  - Output all modified data to stable storage
  - Output a log record <checkpoint> to stable storage

Review Slides (5)

- What is atomic transaction?
- Purpose of commit, abort, rolled-back?
- How to use log and checkpoints?

Textbook Problem Set

- 6.1: The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

  boolean flag[2];/*initiallyfalse*/

  int turn;

The structure of process Pi (i ==0 or 1) is shown in the figure below; the other process is Pj (j ==1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; // do nothing
         flag[i] = true;
      }
   }

      // critical section

   turn = j;
   flag[i] = false;

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

Figure 6.42 The structure of process P_i in Dekker's algorithm.

Textbook Problem Set

- 6.2: Explain why disable interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.
- 6.5: A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.
- 6.9: Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.
- 6.12: Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the TestAndSet() instruction. The solution should exhibit minimal busy waiting.

Backup

Case Study:
Solaris 2
Windows XP

Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments.
  - Mutex and semaphore always serialize data accesses
- Uses **condition variables** and **readers-writers** locks when longer sections of code need access to data.
  - Efficient for data that is accessed frequently, but in a read-only manner

Solaris 2 Adaptive Mutex

- Multiprocessor system
  - Data locked (i.e. in use)
    - Locking thread is running → requesting thread spins on the mutex (spinlock)
    - Locking thread is not in run state → requesting thread blocks on the mutex (waiting lock)
- Uniprocessor system
  - Requesting thread always blocks

Solaris 2 Turnstile

- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - A turnstile is a queue structure containing threads blocked on a lock
- To prevent a priority inversion, turnstiles are organized according to a *priority-inheritance protocol*
  - Temporarily inherit the priority of the high-priority thread (blocked on this lock)

XP Synchronization

- Use **interrupt masks** to protect access to global resources on uniprocessor systems (disable interrupt)
- Uses **spinlocks** on multiprocessor system
- Dispatcher objects: either in signaled or nonsignaled state
  - Signaled: object is available immediately
  - Nonsignaled: object is not available
  - Thread queue associated with each object
  - *WaitForSingleObject* or *WaitForMultipleObjects*