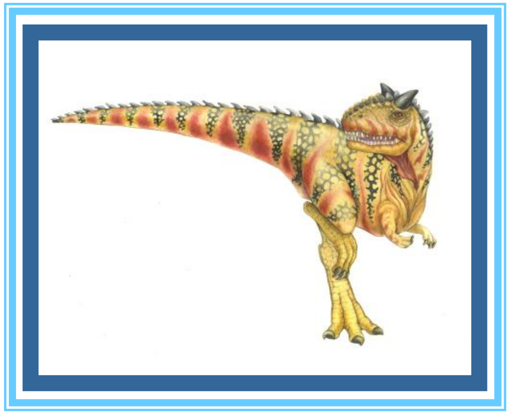
Chapter 5: Process Synchronization



p1 : def f1( )

while 1:

print (A)

print (B)

p2 : def f2( ):

while 1:

print (C)

print (D)

ABCDCD

ACDBABCD

ABCACDBDAB ✓

P1 P2 P

AABB

DCABX

Producer

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

        while (counter == BUFFER_SIZE) ;
              /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
      while (counter == 0)
            ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
        counter--;
      /* consume the item in next consumed
*/
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

S0: producer execute **`register1 = counter`** {register1 = 5}
S1: producer execute **`register1 = register1 + 1`** {register1 = 6}
S2: consumer execute **`register2 = counter`** {register2 = 5}
S3: consumer execute **`register2 = register2 – 1`** {register2 = 4}
S4: producer execute **`counter = register1`** {counter = 6 }
S5: consumer execute **`counter = register2`** {counter = 4}

انحصار متقابل ← منابع مشترک

نرم افزاری

سخت افزاری

OS

programming

پیشرفت

پیشرفت

اگر یکی توی بخش بحرانی هست بقیه انتظار بکشند

گرسنگی ← انتظار محدود

تلاش اول Dekker

✓ انتظار محدود ✗

✗ پیشرفت

✓ انحصار متقابل

```
P0(void)
{
  while(TRUE)
  {
     while( turn != 0) ;  /*wait*/
     critical-section( );
     turn = 1;
     non-critical- section( );
  }
}
```

```
P1(void)
{
  while(TRUE)
  {
     while( turn != 1) ;  /*wait*/
     critical–section( );
     turn = 0;
     non-critical- section( );
  }
}
```

Dekker

True True

```
boolean   flag[2] = {FALSE,FALSE};
```

P0(void){	**P1**(void){
while(TRUE) {	**while**(TRUE) {
while(flag[1]);	**while(** flag[0]);
flag[0] = TRUE;	flag[1] = TRUE;
critical-section( );	**critical-section( );**
flag[0] = FALSE;	flag[1] = FALSE;
non-critical- section ( );	non-critical- section ( );
}	}
}	}

flag0 f T f T

flag1 f T f T

boolean flag[2] = {FALSE,FALSE};

```
P0(void){
  while(TRUE)  {
      flag[0] = TRUE ;
      while( flag[1] );
      critical-section( );
      flag[0]= FALSE;
      non-critical- section( );
    }
}
```

```
P1(void){
  while(TRUE)  {
      flag[1] = TRUE;
      while( flag[0] );
      critical-section( );
      flag[1]= FALSE;
      non-critical- section( );
    }
}
```

Dekker

flag

boolean flag[2] = {FALSE,FALSE};

P0(void)	P1(void)

<pre>{
  while(TRUE)
  {
    flag[0] = TRUE;
    while(flag[1])
    {
      flag[0] = FALSE;
      delay_for_a_short_time( );
      flag[0] = TRUE;
    }
    critical-section( );
    flag[0] = FALSE;
    non-critical- section( );
  }
}</pre>
	<pre>{
	  while(TRUE)
	  {
	    flag[1] = TRUE;
	    while(flag [0])
	    {
	      flag[1] = FALSE;
	      delay_for_a_short_time( );
	      flag[1] = TRUE;
	    }
	    critical-section( );
	    flag[1] = FALSE;
	    non-critical- section( );
	  }
	}</pre>

Dekker

بن بست

Live lock

انتظار محدود

بن بست

انحصار متقابل

پیشرفت

Dekker

```
boolean  flag[2] = {FALSE,FALSE};
turn=0;
```

```
P0(void){
  while(TRUE){
      flag[0] = TRUE;
      while( flag[1] )
       if (turn == 1){
          flag[0] = FALSE;
          while( turn==1) do;
          flag[0] = TRUE;
       }
     critical-section( );
      turn = 1;
      flag[0] = FALSE;
      non-critical-section ( );
  }
}
```

```
P1(void){
  while(TRUE){
    flag[1] = TRUE;
    while( flag[0])
       if (turn == 0){
          flag[1] = FALSE;
          while( turn==0) do;
          flag[1] = TRUE;
       }
    critical-section( );
     turn = 0;
     flag[1] = FALSE;
     non-critical-section( );
  }
}
```

تلاش های Decker	انحصار متقابل	پیشرفت	انتظار محدود
تلاش اول	✓	–	✓
تلاش دوم	–	✓	–
تلاش سوم	✓	✓	–
تلاش چهارم	✓	✓	-
تلاش پنجم	✓	✓	✓

Peterson

```
boolean   flag[2] = {FALSE,FALSE};
turn=0;
```

flag 0   flag 1

P0	P1
P0(void){	**P1**(void){
{	{
while (TRUE){	**while** (TRUE){
flag[0] = TRUE;	flag[1] = TRUE;
turn = 0;	turn = 1;
while (turn==0 **&&** flag[1]	**while**(turn==1 **&&** flag[0]
);	**);**
critical-section( );	**critical-section( );**
flag[0] = FALSE;	flag[1] = FALSE;
non-critical-section( );	non-critical-section( );
}	}
}	}

راه حل سخت‌افزاری:

با متوقف سازی وقفه

```
disable_interrupts()

critical section()

enable_interrupts()
```

✓ انحصار حاصل

با کمک سخت‌افزاری

tsl → اتمی

test & set lock

انحصار متقابل

مشکل دارد

```
enter_region:
  tsl    reg , lock
  cmp    reg , #0
  jne    enter_region
  ret
```

```
enter_region:
  move   reg , #1
  swap   reg, lock
  cmp    reg, #0
  jne    enter_region
  ret
```

tsl
swap

سخت‌افزاری

دودویی

شمارشی

Semaphore

زبان برنامه‌نویسی

wait( )

Signal( )

Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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

Readers-Writers Problem (Cont.)

- The structure of a reader process

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

Producer-Consumer

۱– سمافور mutex :

سمافوری برای رعایت شرط انحصار متقابل است، تا تولید کننده و مصرف کننده بـه طـور همزمـان بـه بـافر دسترسی نداشته باشند. (با مقدار اولیه 1)

۲– سمافور full :

سمافوری برای شمارش تعداد خانه های پر بافر (با مقدار اولیه 0)

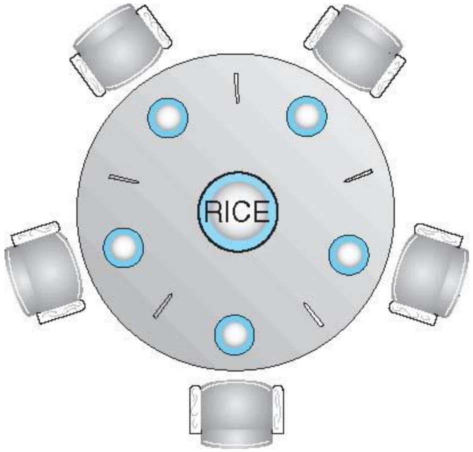
۳– سمافور empty :

سمافوری برای شمارش تعداد خانه های خالی بافر (با مقدار اولیه n)

```
void  producer (void){
   int   item;
   while(TRUE)      {
        item= produce( );
        wait(empty);
        wait(mutex);
        insert(item);
        signal(mutex);
        signal(full);     }
}
```

```
void  consumer(void){
    int  item;
    while(TRUE)     {
       wait(full);
       wait(mutex);
       item=remove( );
       signal(mutex);
       signal(empty);
       consume( );      }
}
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

                  //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

                   //  think

} while (TRUE);
```

- What is the problem with this algorithm?

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example
  - semaphores *A* and *B*, initialized to 1

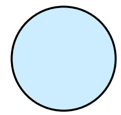
P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Dining-Philosophers Problem Algorithm

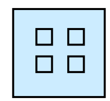
```
semaphore  room=4;
semaphore  fork[5]={1};
void   philosopher (int  i){
    while(TRUE){
        think( );
        wait( room );
        wait( fork[i] );
        wait( fork[(i+1) % 5] );
        eat( );    ناحیه بحرانی
        signal ( fork[(i+1) % 5] );
        signal ( fork[i] );
        signal ( room );
    }
}
void  main( ){
  parbegin (p(0),p(1),p(2),p(3),p(4)));
}
```

Resource-Allocation Graph (Cont.)

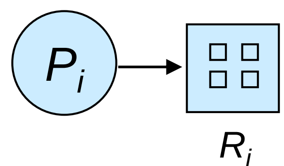
- Process



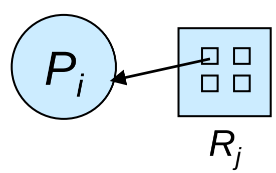
- Resource Type with 4 instances



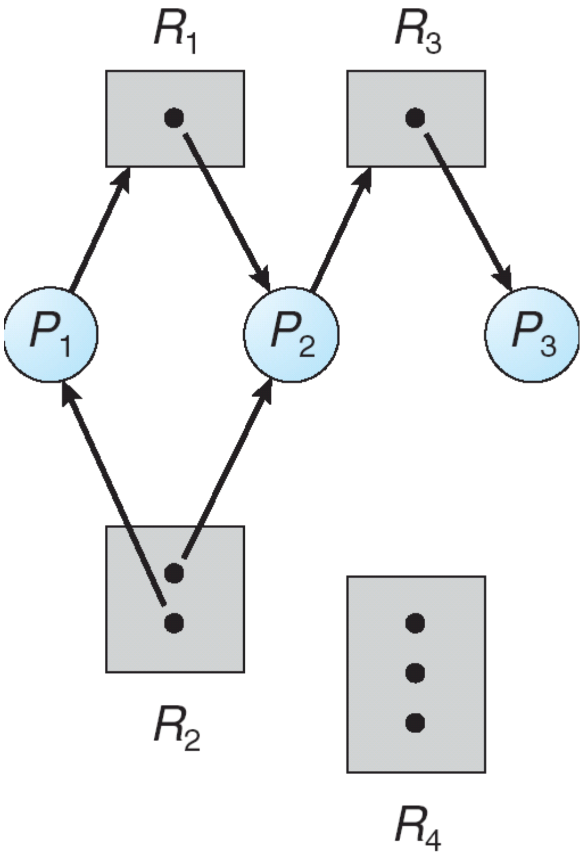
- P_i requests instance of R_j



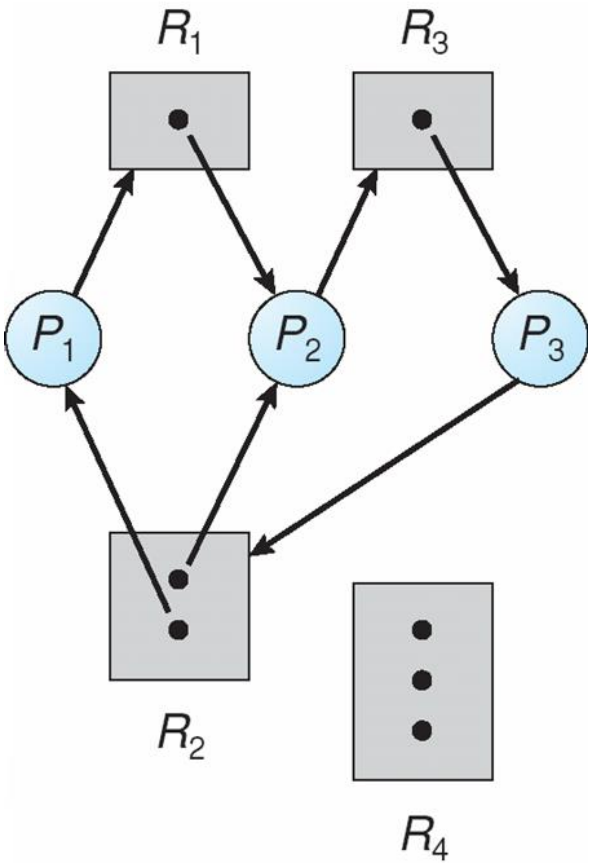
- P_i is holding an instance of R_j



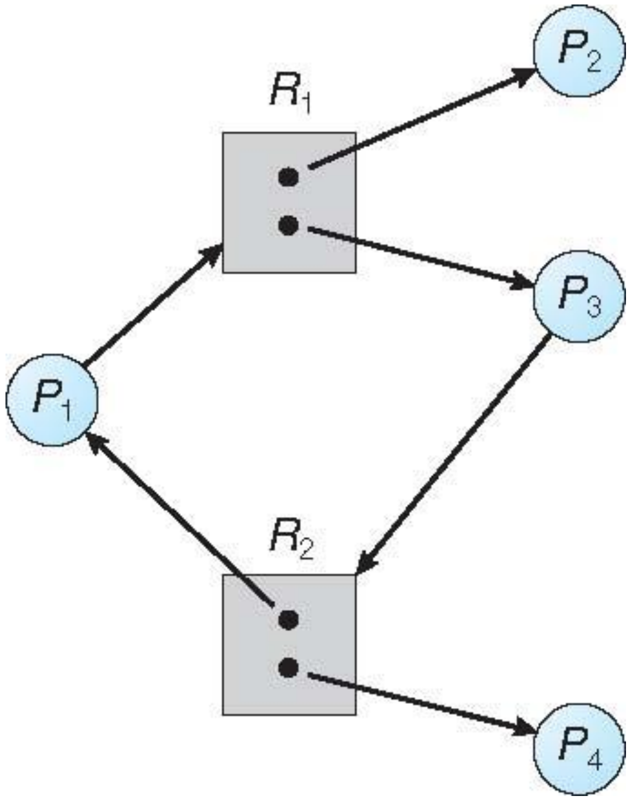
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

End of Chapter 5