Operating System Principles

操作系统原理

Inter-Process Communication

李旭东

leexudong@nankai.edu.cn

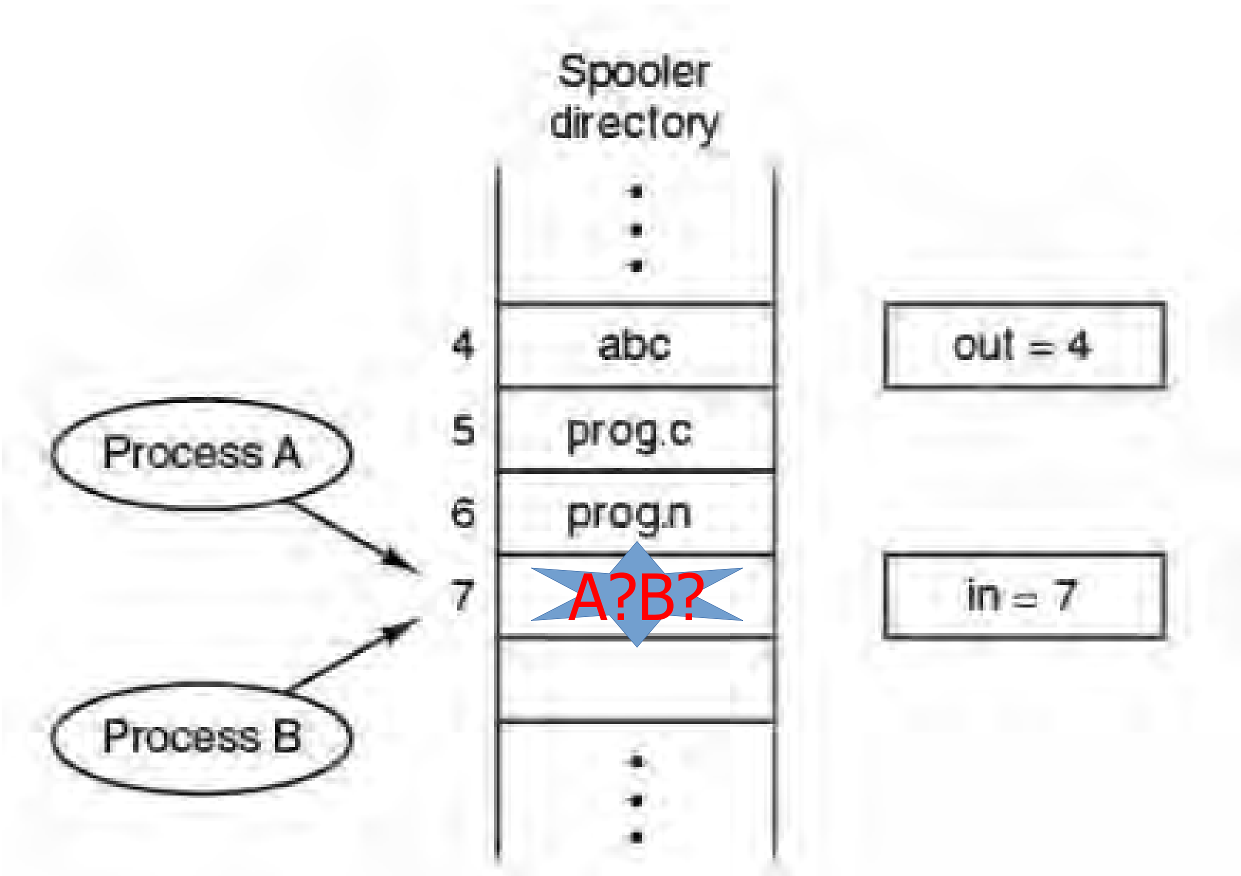
Nankai University

Objectives

- race condition
- mutual exclusion
- synchronization
- communication

Race Condition

- race condition 竞态



Murphy’s law

If something can go wrong, it will

Race Conditions 竞态

two or more processes are reading or writing some shared data and the final result depends on who runs precisely

Processes Relations

- Resource sharing
- Co-operation
- …

Critical Regions

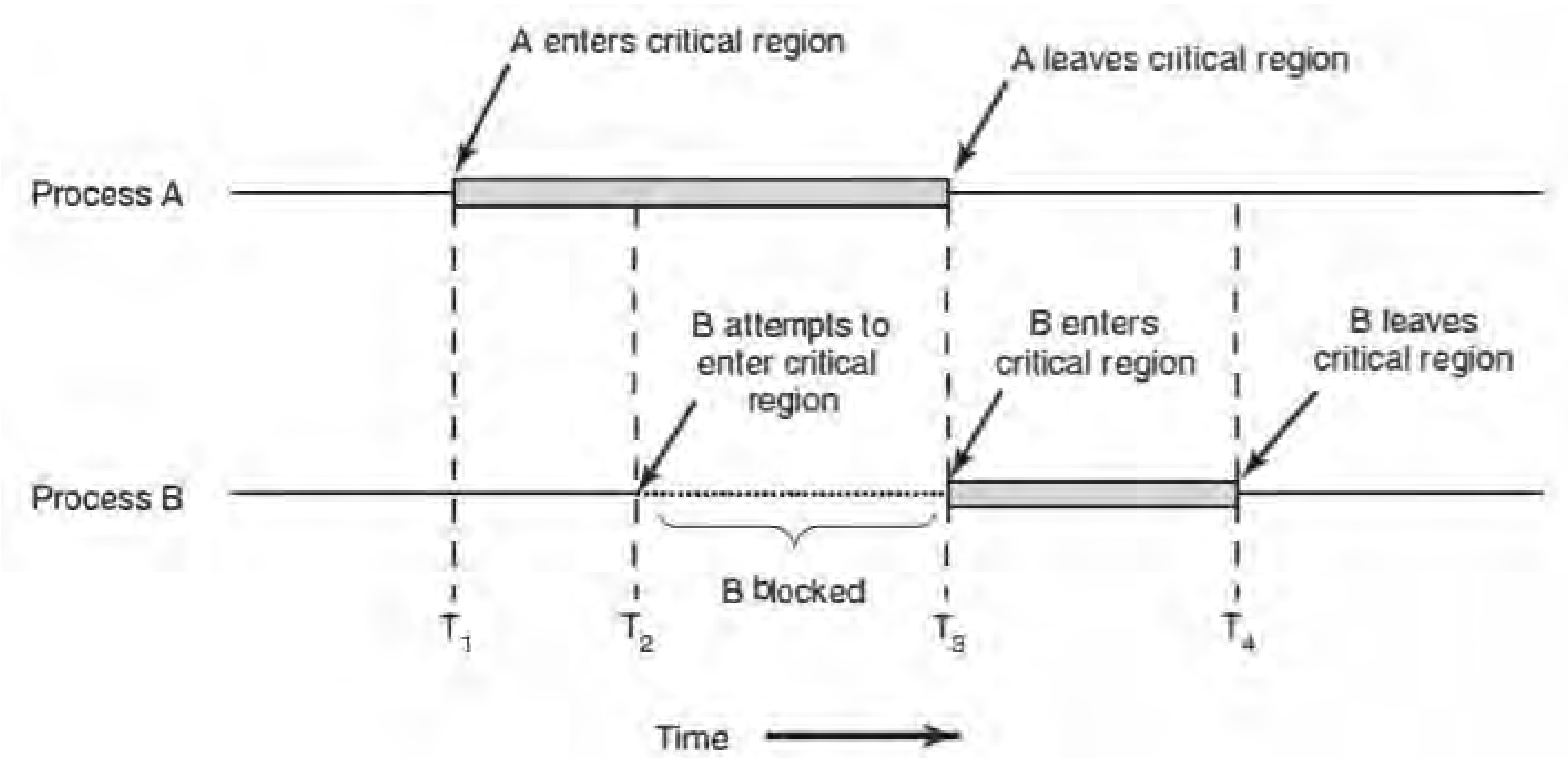
- Critical Resource 临界资源
  - File, memory, semaphore, ...
- Critical Regions 临界区
  - Sometimes a process has to access shared memory or files, or do other critical things that can lead to races
  - That part of the program where the shared memory is accessed is called the critical region or critical section

Four Prerequisites 前提
for avoiding race conditions

- **1. No two processes may be simultaneously inside their critical regions.**
- **2. No assumptions may be made about speeds or the number of CPUs.**
- **3. No process running outside its critical region may block other processes.**
- **4. No process should have to wait forever to enter its critical region.**

Mutual Exclusion

- 互斥访问



Mutual Exclusion

- How to access critical resource

Repeat

entry section

Critical sections;

exit section

Remainder section;

Until false

Solutions of Mutual Exclusion

- Disabling Interrupts
  - With interrupts disabled, no clock interrupts can occur.
  - The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to an other process.
  - Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

Solutions of Mutual Exclusion

- Lock Variables
  - a software solution
  - Consider having a single, shared (lock) variable, initially 0.
  - When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0.
  - Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

?Race Condition

Solutions of Mutual Exclusion

- Strict alternation 严格轮换法
  - the integer variable turn, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Strict alternation (cont.,)

- Problem 1
  - busy waiting

    Continuously testing a variable until some value appears
- Problem 2
  - running speed does not match
- Problem 3
  - process 0 is being blocked by a process not in its critical region
- Case: spinlock 自旋锁

Solutions of Mutual Exclusion

- Peterson's Solution

```
#define FALSE 0
#define TRUE  1
#define N       2    /*number of processes*/
shared int turn;                /*whose turn is it?*/
shared int interested[N];  /*all values initially 0*/
void enter_region(int process)
{
    int other;
    other=1-process;
    interested[process]=TRUE;
     turn=process;
?while(turn!=process && interested[other]==TRUE);   Is ok?
}
void leave_region(int process)
{
     interested[process]=FALSE;
}
```

Solutions of Mutual Exclusion

- TSL Instruction
  - test and set lock

TSL REGISTER,LOCK

```
enter_region:
    TSL REGISTER,LOCK1
    CMP  REGISTER,#0
    JNE   enter_region
    RET

leave_region:
    MOVE LOCK1,#0
    RET
```

Solutions of Mutual Exclusion

- TSL Instruction
  - test and set lock TSL REGISTER,LOCK
  - It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock.
  - The operations of reading the word and storing into it are guaranteed to be indivisible-no other processor can access the memory word until the instruction is finished.

Solutions of Mutual Exclusion

- Swap Instruction
  - XCHG

```
Procedure Swap(var a,b:boolean)
Var temp:boolean;
Begin
  temp:=a;
  a:=b;
  b:=temp;
End;
```

Solutions of Mutual Exclusion

- Summary
  - busy waiting
    - Spinning
      - a process repeatedly checks to see if a condition is true
- priority inversion problem
  - a high priority task is indirectly preempted by a medium priority task effectively "inverting" the relative priorities of the two tasks
    - The higher priority task: enough time, but no critical resource
    - The lower priority task: using critical resource, but no time

New Solution of Mutual Exclusion

- Semaphore
  - 1965, E. W. Dijkstra (迪科斯彻)
  - A semaphore, as defined in the dictionary, is a mechanical signalling device or a means of doing visual signalling
  - The analogy typically used is the railroad mechanism of signaling trains, where mechanical arms would swing down to block a train from a section of track that another train was currently using. When the track was free, the arm would swing up, and the waiting train could then proceed.

New Solution of Mutual Exclusion

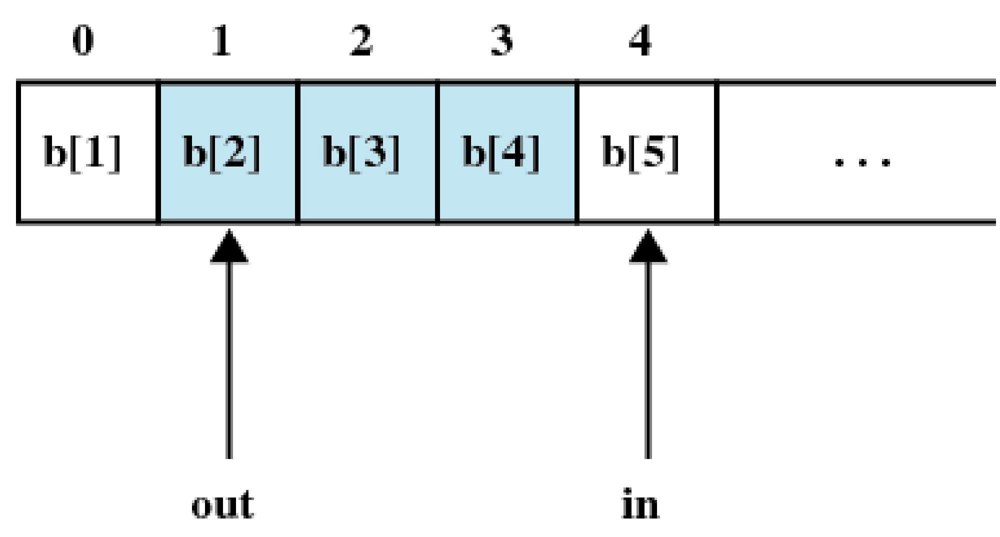
- Semaphore
  - A variable
  - Has two atomic operations: Primitive 原语
    - Down, Up
    - P (Proberen), V (Verhagen); Wait(), Signal()
  - Down()
    - If so, it decrements the value and just continues. If the value is 0, the process is put to sleep without completing the down for the moment.
  - Up()
    - The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system and is al lowed to complete its down

IPC Primitives 原语

- inter-process communication primitives
  - block in stead of wasting CPU time when they are not allowed to enter their critical regions
- example
  - sleep
  - wakeup

The Producer-Consumer Problem

- The Producer-Consumer Problem(PCP)
  - the bounded-buffer problem
  - multi-processes share a common, fixed-size buffer



First Solution: The Producer-Consumer Problem

? Race Condition

Add: wakeup waiting bit

```
#define  N 100
/*number of slots in the buffer*/
int count=0;
/*number of items in the buffer*/
void producer(void){
  int item;
  while(TRUE){
    produce_item(&item);
    if (count==N) sleep();
    enter_item(item);
    count=count+1;
    if (count==1)
       wakeup(consumer);
  }
}
```

```
void comsumer(void){
   int item;
   while(TRUE){
     if(count==0) sleep();
     remove_item(&item);
     count=count-1;
     if (count==N-1)
        wakeup(producer);
     consume_item(item);
   }
}
```

PCP using Semaphore

```
#define  N 100     /*number of slots in the buffer*/
typedef int semaphore;
semaphore mutex=1;
semaphore empty=N;
semaphore full=0;
void producer(void){
    int item;
    while(TRUE){
    produce_item(&item);
    down(&empty);
    down(&mutex);
    enter_item(item);
    up(&mutex);
    up(&full);
    }
}
```

```
void consumer(void){
    int item;
    while(TRUE){
        down(&full);
        down(&mutex);
        remove_item(&item);
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
```

? exchange two lines

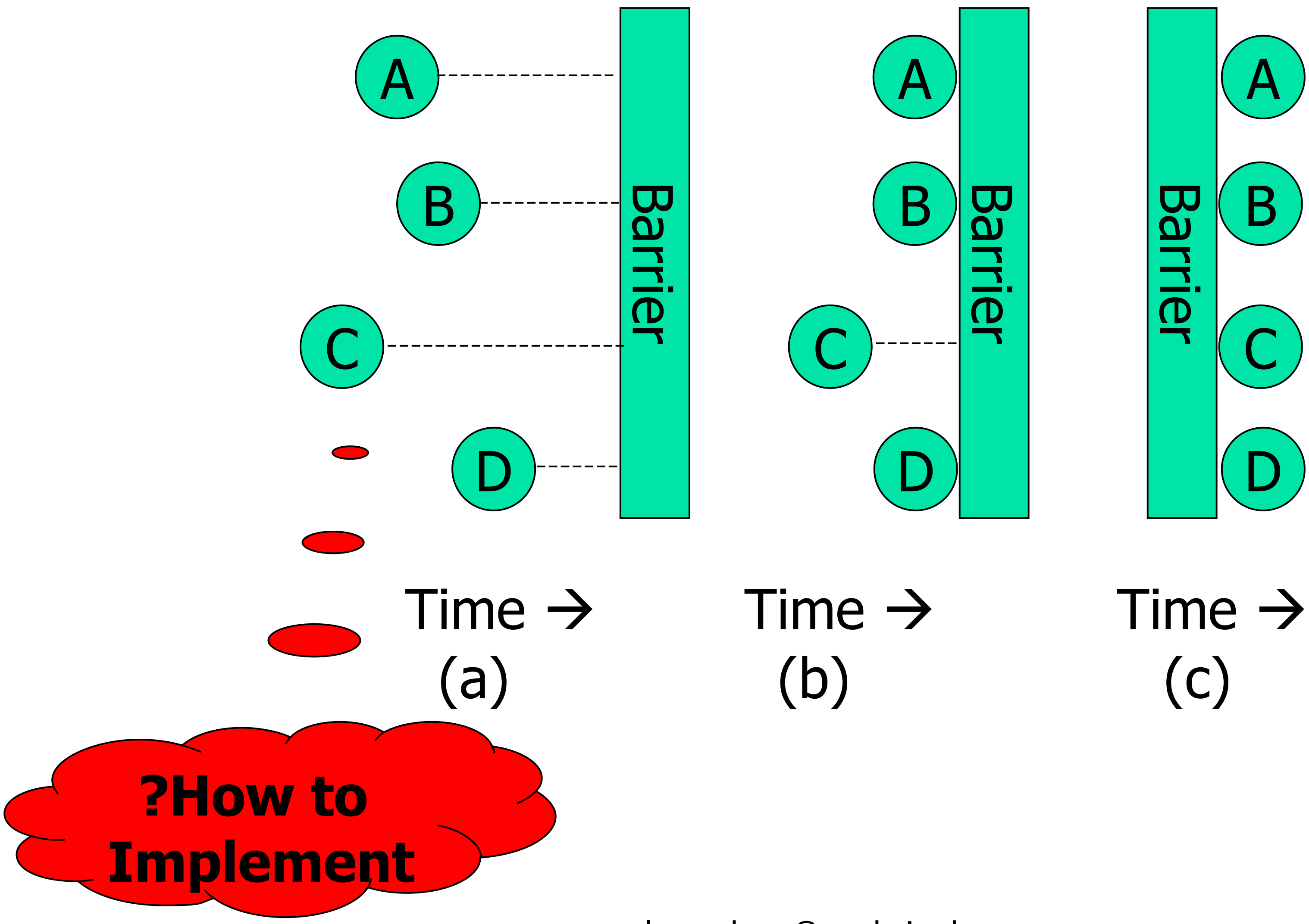
down(&mutex);

down(&empty);

Semaphore Scenarios

- mutual exclusion
  - semaphore mutex=1;
- synchronization
  - semaphore empty=N;
  - semaphore full=0;

Barrier(屏障)



Implement of mutex

- Mutex
  - Value: 0,1

```
mutex_lock:
        TSL REGISTER,MUTEX
        CMP REGISTER,#0
        JZE  ok
        CALL thread_yield           ;schedule another thread
        JMP mutex_lock
ok: RET

mutex_unlock:
        MOVE MUTEX,#0
        RET
```

Pthread Calls: Semaphores

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Semaphore Set

- Why

Process A:
  wait(mutex1);
  wait(mutex2);

Process B:
  wait(mutex2);
  wait(mutex1);

- Semaphore Set

  Swait(S1,S2,…,Sn)

  Ssignal(S1,S2,…,Sn)

Condition Variables

- Condition Variable
  - typedef boolean cond;
    - true,false
    - Using semaphore
- Functions

Condition_init(cond)

condition_wait(cond,mutex)

condition_signal(cond)

condition_broadcast(cond)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                       /* put item in buffer */
                pthread_cond_signal(&condc);      /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                       /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                       /* take item out of buffer */
                pthread_cond_signal(&condp);      /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

Example: condition variables

```
int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Event Counter

- Reed and Kanodia,1979
- Three operations
  - Read(E)
  - Advance(E)
    - E=E+1 (atomic)
  - Await(E,v)
    - Wait until E>=v

PCP:Event Counter

```
#define  N 100
typedef int event_counter;
event_counter in =0;
event_counter out =0;
void producer(void){
   int item,sequence=0;
   while(TRUE){
     produce_item(&item);
     sequence=sequence+1;
     await(out,sequence-N);
     enter_item(item);
     advance(&in);
   }
}
```

```
void consumer(void){
        int item,sequence=0;
        while(TRUE){
          sequence=sequence+1;
          await(in,sequence);
          remove_item(&item);
          advance(&out);
          consume_item(item);
        }
}
```

Monitor

- Some errors with using semaphore
  - Error 1:

signal(mutex)

critical section

wait(mutex)

  - Error 2:

wait(mutex)

critical section

wait(mutex)

  - Error 3:

ignore wait(mutex)

  - Error 4:

ignore signal(mutex)

Monitor 管程

Hoare(1974),Brinch Hansen(1975)

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .     .     .
    end;
end monitor;
```

Monitor

- Some errors with using semaphore
  - Error 1:

signal(mutex)

critical section

wait(mutex)

  - Error 2:

wait(mutex)

critical section

wait(mutex)

  - Error 3:

ignore wait(mutex)

  - Error 4:

ignore signal(mutex)

PCP: Monitor

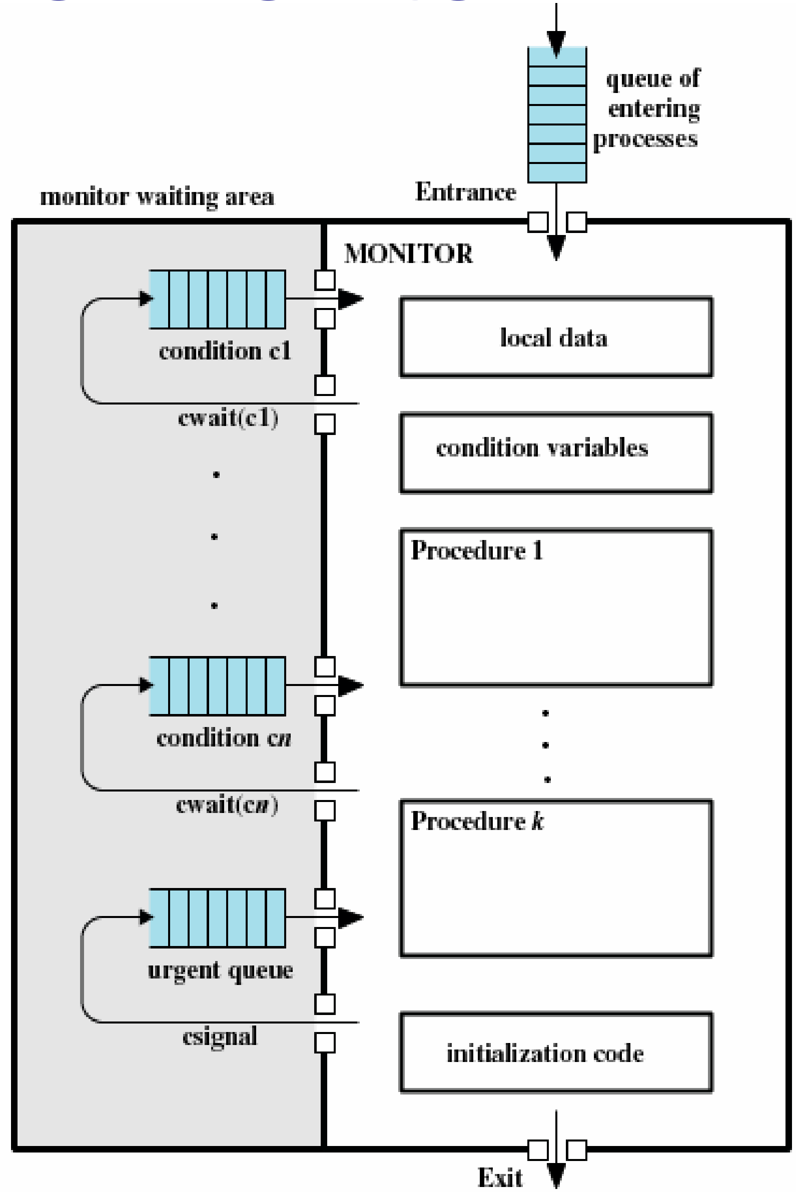
```
monitor ProducerConsumer
    condition full,empty;
    integer count;
    procedure insert(item:integer);
    begin
    if count=N then wait(full);
    insert_item(item);
    count:=count+1;
    if count=1 then signal(empty);
    end;
    function remove:integer;
    begin
    if count=0 then wait(empty);
    remove=remove_item;
    count:=count-1;
    if count=N-1 then signal(full);
    end;
    count:=0;
end monitor;
```

```
procedure producer;
begin
  while true do
     begin
        item=produce_item;
        ProducerConsumer.insert(item);
     end
end;
procedure consumer;
begin
  while true do
  begin
     item=ProducerConsumer.remove;
     consume_item(item);
  end
end;
```

Implement of Monitor

```
type monitor-name=monitor
  variable declarations
procedure entryP1(…);
begin
  …
end;
procedure entryP2(…);
begin  …  end;
…
procedure entryPn(…);
begin  …  end;

begin
   initialization code
end
```



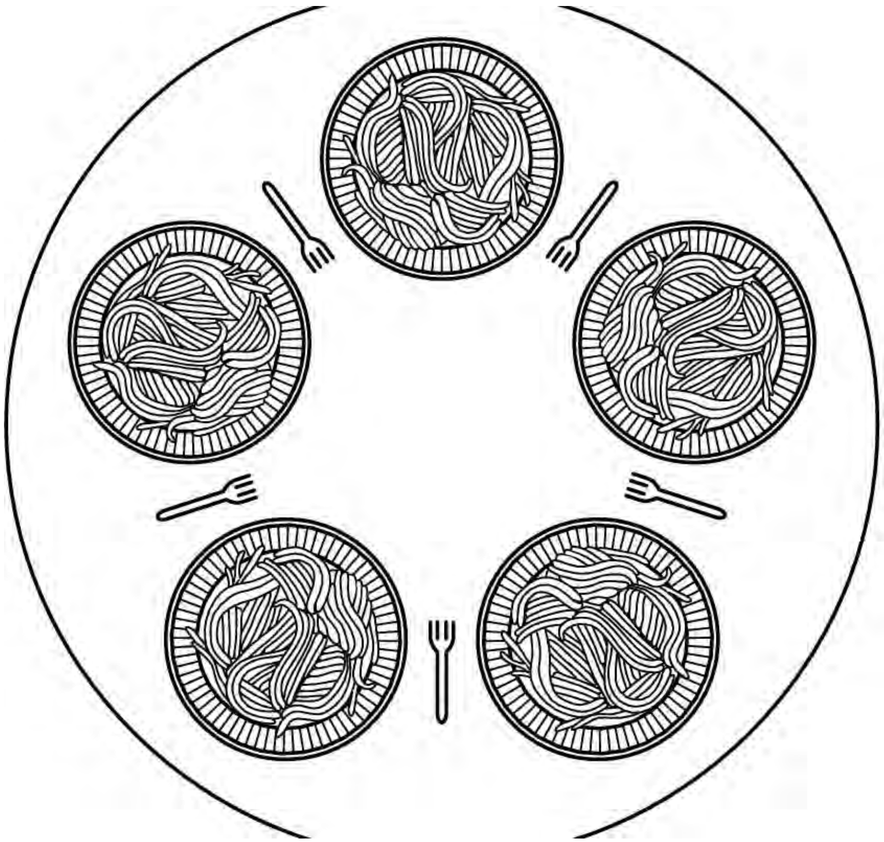
Classical IPC Problems

- The Dining Philosophers Problem(DPP)*
- The Readers and Writers problem(RWP)*
- The Sleeping Barber Problem(SBP)*

The Dining Philosophers Problem

1965, Dijkstra

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock



The Dining Philosophers Problem

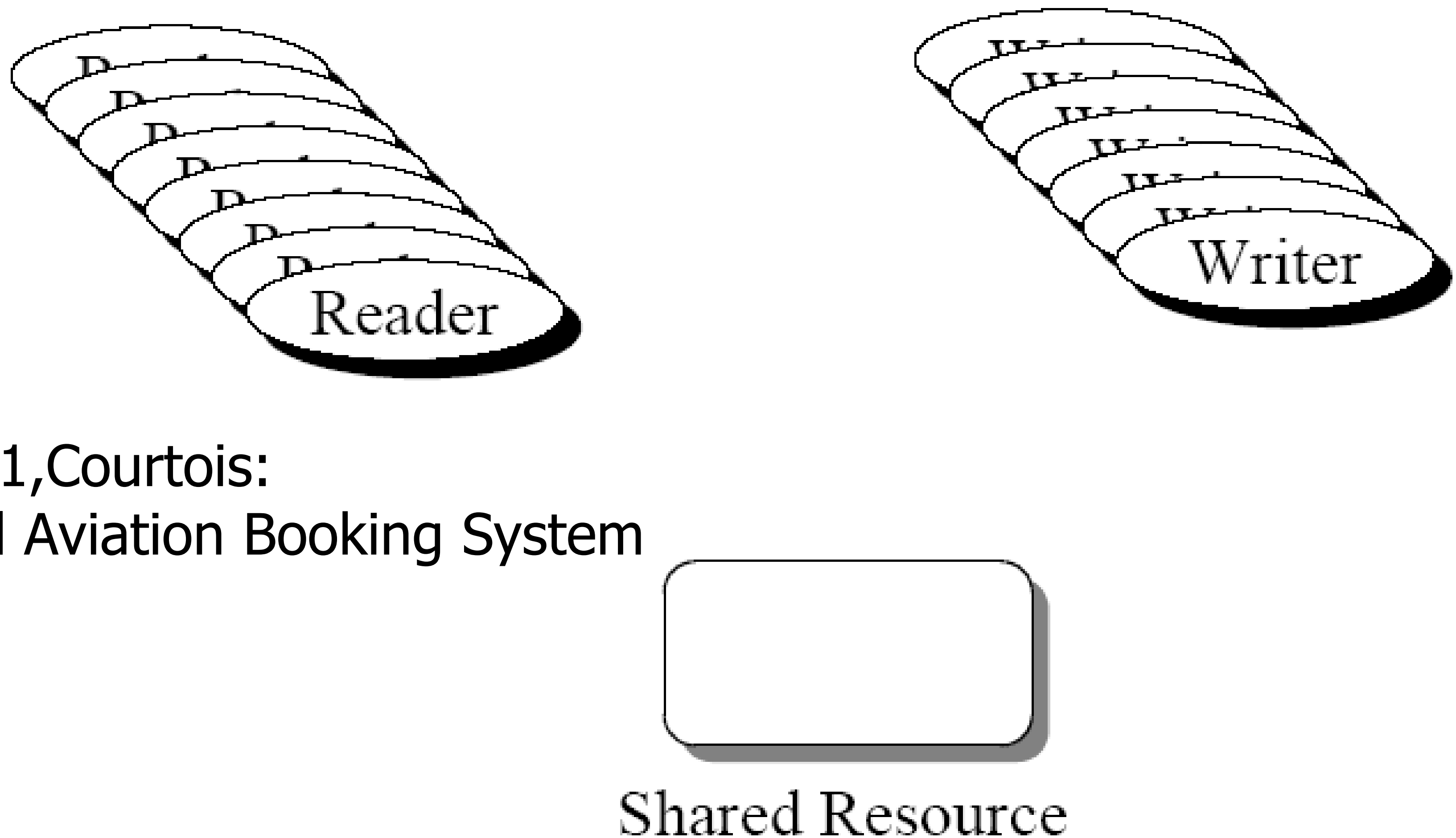
```
#define N  5
void philosopher(int i){
  while(TRUE){
      think();
      take_fork(i);
      take_fork((i+1)%N);
      eat();
      put_fork(i);
      put_fork((i+1)%N);
  }
}
```

The Dining Philosophers Problem

```
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING  0
#define  HUNGRY  1
#define  EATING    2
typedef int semaphore;
int state[N];
semaphore mutex=1;
semaphore s[N];
void philosopher(int){
  while(TRUE){
     think();
       take_forks(i);
     eat();
        put_forks(i);
  }
}
```

```
void take_forks(int i){
  down(&mutex);
  state[i]=HUNGRY;     test(i);
  up(&mutex);
  down(&s[i]);
}
void put_forks(int i){
  down(&mutex);
  state[i]=THINKING;
  test(LEFT);     test(RIGHT);
  up(&mutex);
}
void test(int i){
  if(state[i]==HUNGRY && state[LEFT]!=EATING
     && state[RIGHT]!=EATING){
    state[i]=EATING;
    up(&s[i]);
  }
}
```

The Readers and Writers problem



1971,Courtois:
Civil Aviation Booking System

RWP: Solution 1

```
typedef int semaphore;
semaphore  mutex=1;
semaphore  db=1;
int rc=0;
void  reader(void){
   while(TRUE){
      down(&mutex);
      rc=rc+1;
      if(rc==1) down(&db);
      up(&mutex);
      read_data_base();
       down(&mutex);
      rc=rc-1;
      if (rc==0) up(&db);
      up(&mutex);
      use_data_read();
}}
```

```
void  writer(void){
   while(TRUE){
       think_up_data();
       down(&db);
       write_data_base();
       up(&db);
   }
}
```

```
program readersandwriters;
var readcount,writecount:integer;
x,y,z,wsem,rsem:semaphore(:=1);
procedure reader;
begin
    repeat
    wait(z);
    wait(rsem);
    wait(x);
    readcount:= readcount+1;
    if readcount=1 then wait(wsem);
    signal(x);
    signal(rsem);
    signal(z);
    READUNIT;
    wait(x);
    readcount:= readcount-1;
    if readcount=0 then signal(wsem);
    signal(x);
    forever
end;
```

z 是否必须？

```
procedure writer;
begin    repeat
    wait(y);
    writecount:= writecount+1;
    if writecount=1 then wait(rsem);
    signal(y);
    wait(wsem);
    WRITEUNIT;
    signal(wsem);
    wait(y);
    writecount:= writecount-1;
    if writecount=0 then signal(rsem);
    signal(y);
    forever
end;
begin    readcount,writecount:=0;
    parbegin
    reader;writer;
    parend
end.
```

The Sleeping Barber Problem

The Sleeping Barber Problem

```
#define CHAIRS  5
typedef int semaphore;
semaphore customers=0;
semaphore barbers=0;
semaphore mutex=1;
int waiting =0;
void barber(void){
    while(TRUE){
        down(&customers);
    down(&mutex);
    waiting=waiting-1;
    up(&barbers);
    up(&mutex);
    cut_hair();
   }
}
```

```
void  customer(void){
  down(&mutex);
  if(waiting<CHAIRS){
    waiting=waiting+1;
    up(&customers);
    up(&mutex);
    down(&barbers);
    get_haircut();
  }else{
     up(&mutex);
  }
}
```

Synchronization Approaches: Summary

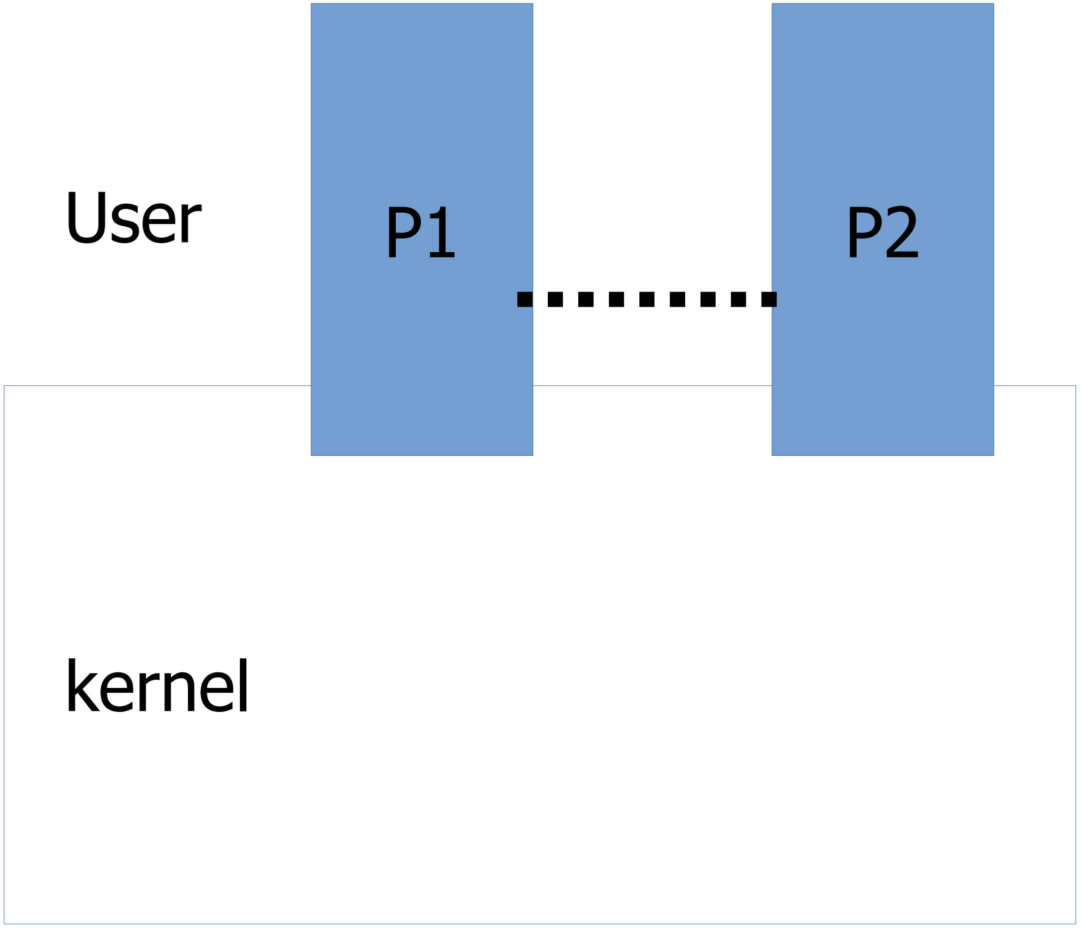
- Mutex
- Semaphore
- Semaphore Set
- Condition Variable
- Event Counter
- Monitor
- Message Passing

All are equivalent!

Alternative Synchronization Approaches

- Transactional Memory
- Functional Programming Languages
- ...

Inter-Process Data Communication

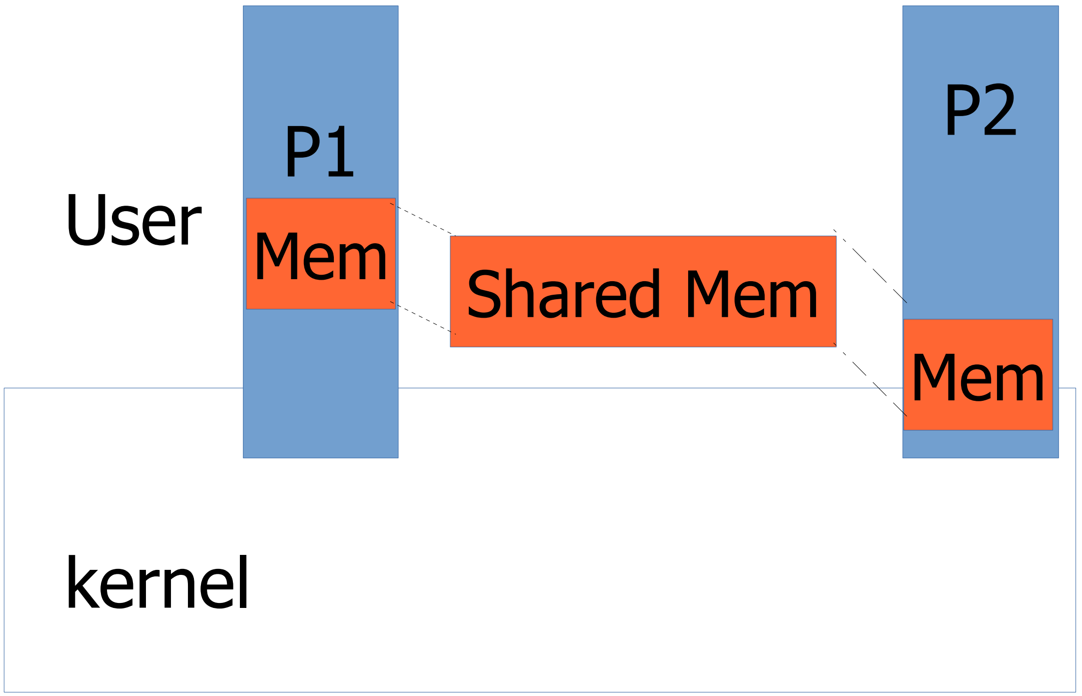


Inter-Process Data Communication

- Shared-Memory System
- Message passing System
- Pipeline System
- Remote Procedure Call

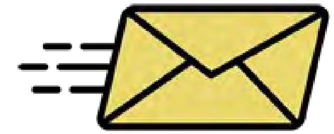
Shared-Memory System

- memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies



Message passing System

- Send(destination, &message)
- Receive(source, &message)



Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queuing Discipline
- FIFO
- Priority

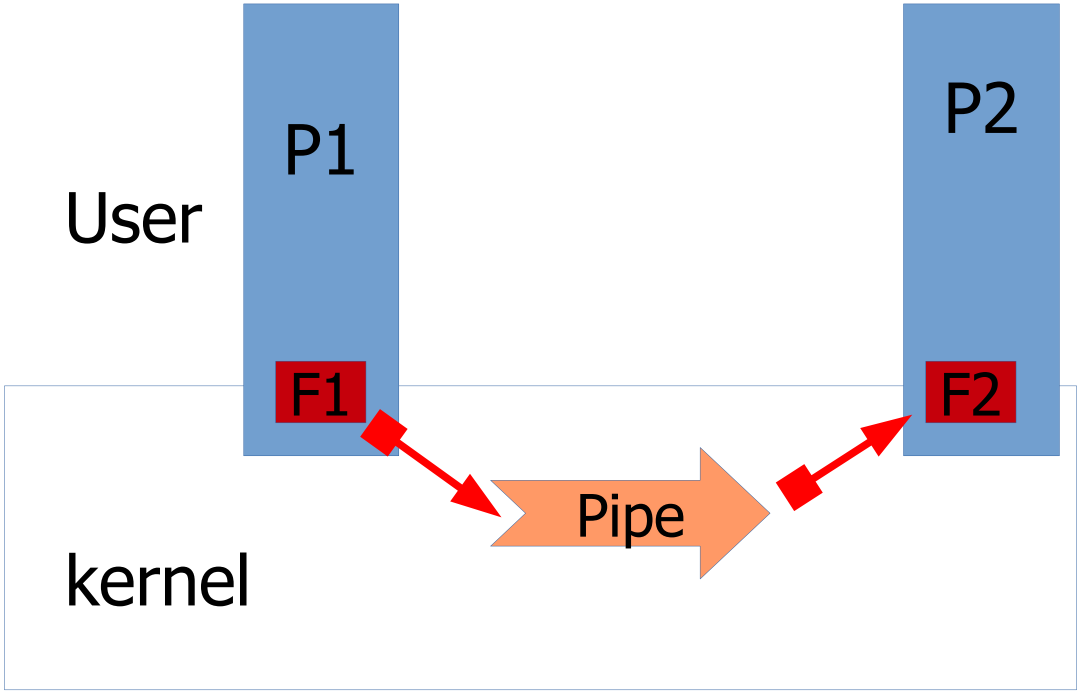
PCP: Message Passing

```
#define  N  100
void producer(void){
   int item;
   message m;
   while(TRUE){
      item=produce_item();
      receive(consumer,&m);
      build_message(&m,item);
      send(consumer,&m);
   }
}
```

```
void consumer(void){
  int item,i;
  message m;
  for(i=0;i<N;i++)
    send(producer,&m);
  while(TRUE){
     receive(producer,&m);
     item=extract_item(&m);
     send(producer,&m);
     consume_item(item);
  }
}
```

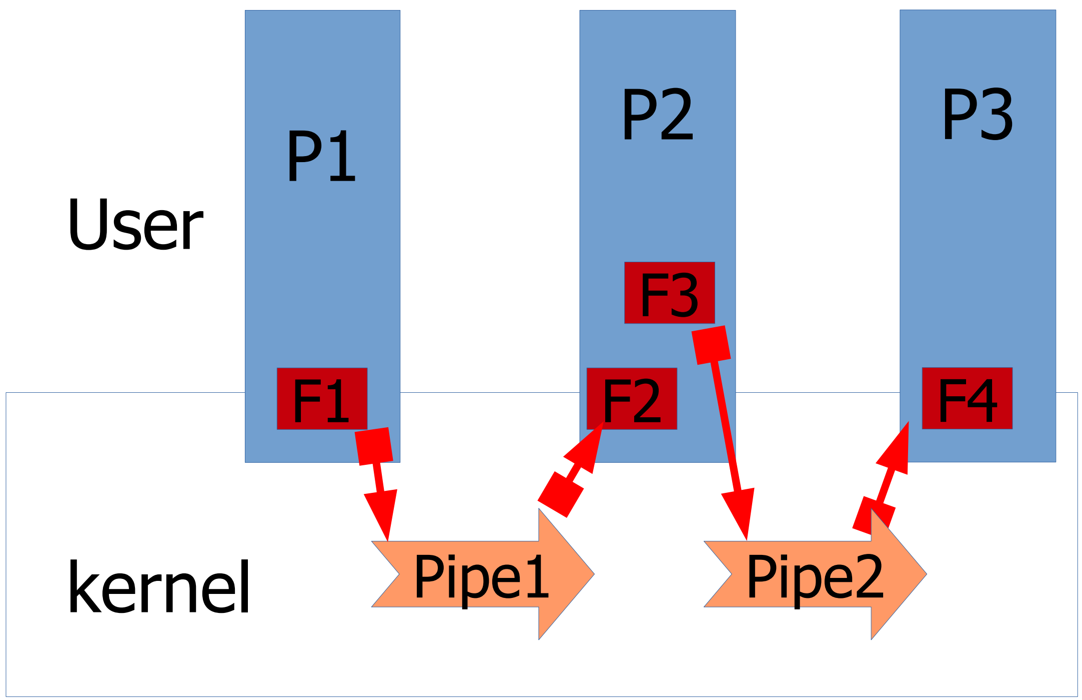
Pipe System

- a set of data processing elements connected in series, where the output of one element is the input of the next one

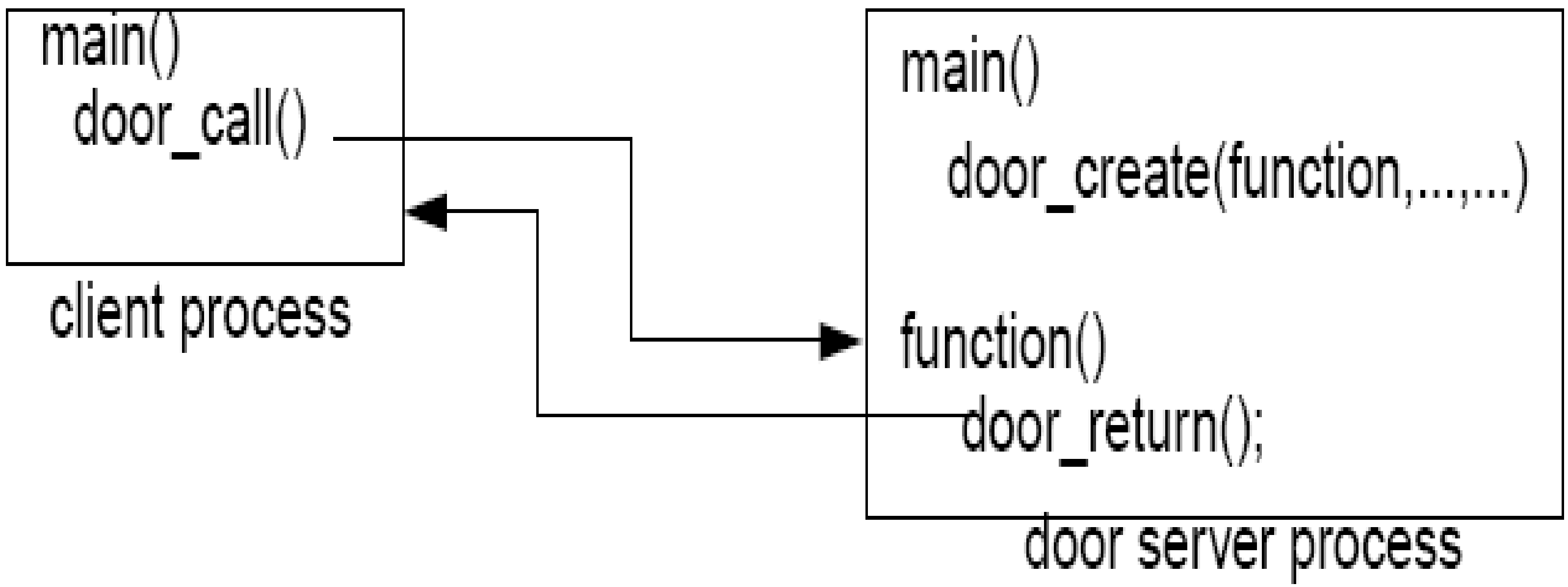


Pipeline System

cat file1 file2 file3 | grep “root” | wc -l



Remote Procedure Call



Summary

- race condition
- mutual exclusion
- synchronization
- communication