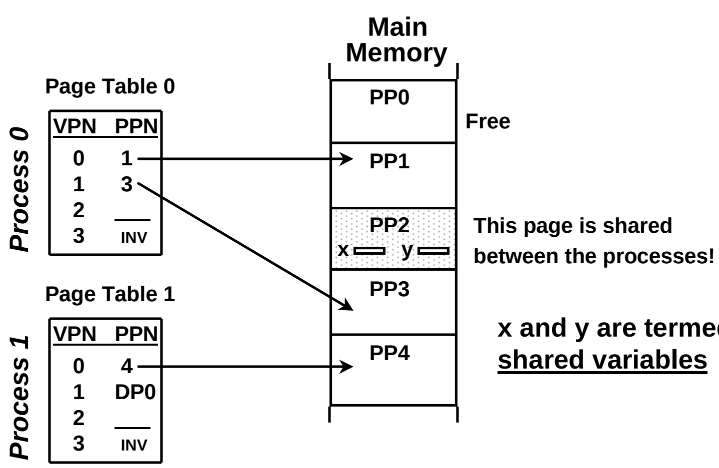
Interprocess Synchronization

1

Recall Memory Protection

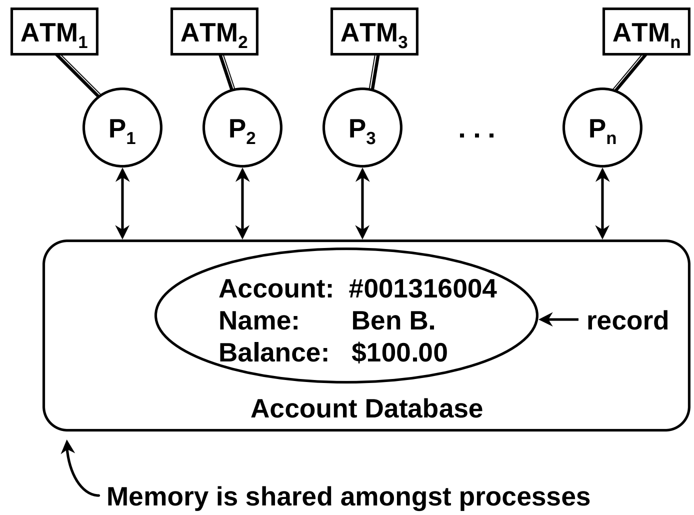


x and y are termed <u>shared variables</u>

2

6004 Spring 1998: L45

Hacking a Fleet ATM



3

Process Routines

Each process has routines to credit and debit balances

<u>Deposit</u> (Acctno, amt)

Balance[Acctno] = Balance[Acctno] + amt ;

<u>Withdrawal</u> (Acctno, amt)

if (Balance[Acctno] $\geqslant$ amt) then

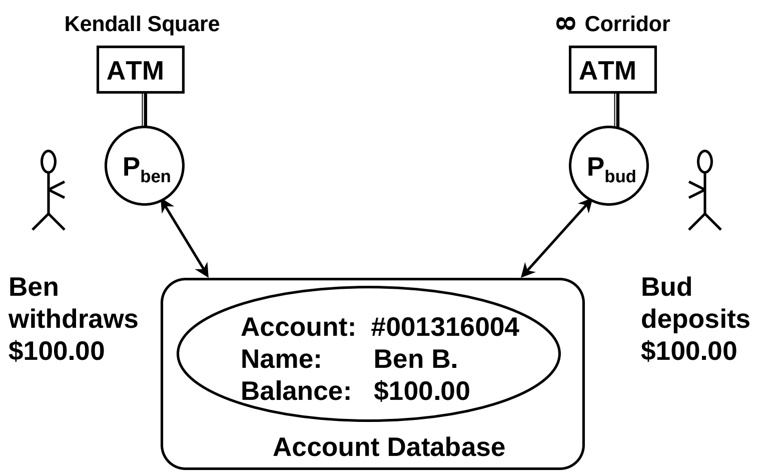
Balance[Acctno] = Balance[Acctno] - amt

else

Display "Call home ASAP!" ;

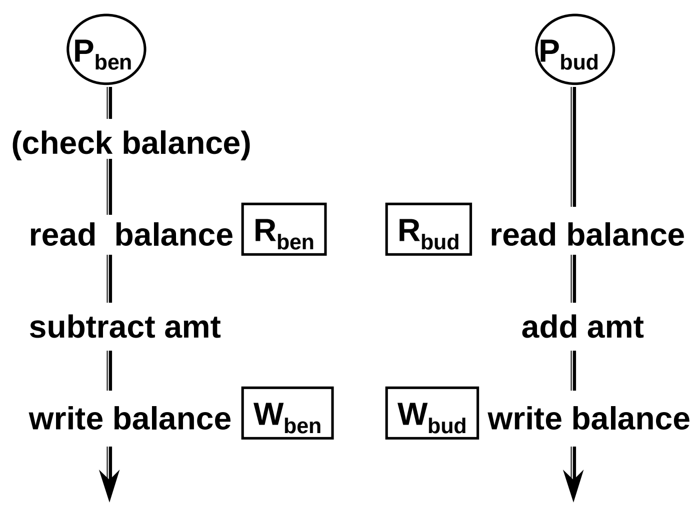
4

A Scam !



5

Scam (contd.)



6

Observations

- Reads and writes from a given process happen in the given "program order"
- But process execution may be ___________________________.
- How many permutations?

	Resulting Balance
– R_{ben} W_{ben} R_{bud} W_{bud}	$100.00
– R_{bud} W_{bud} R_{ben} W_{ben}	________
– R_{ben} R_{bud} W_{bud} W_{ben}	________
– R_{ben} R_{bud} W_{ben} W_{bud}	________

7

Scam Prevention

- The time between a read R_i of a shared variable v in a process and its corresponding write W_i is called a <u>critical section</u>
- Idea: Exclude other processes from accessing the shared variable v while some process in its critical section
- Create another shared variable with each account. Set it to indicate that your process wants to update the account.

8

6004 Spring 1998: L45

Counting Semaphores

Let S be a shared integer semaphore initialized to some value.

wait(S) : **Delays process until count in S > 0. Then decrement S (subtract 1) and proceed.**

signal(S) : **Increment count in S and proceed. Side effect is to allow processes waiting on S to proceed.**

9

Binary Semaphore a.k.a. Mutex a.k.a. Lock

Ben B.

Balance = $100.00

S = 1 → **1 means free**
0 means busy

P_{ben}	**P_{bud}**
wait(S)	**wait(S)**
…	**…**
critical section (C. S.) **Withdrawal**	**critical section (C. S.)** **Deposit**
…	**…**
signal(S)	**signal(S)**

10

Binary Semaphore Signal/Wait Routines

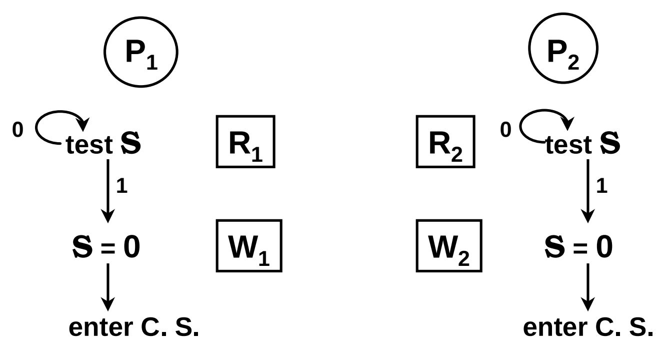
```
signal ( S)
    S = S + 1 ;


wait ( S)
    try_again:  if ( S > 0 ) then          ( if S is 1 )
                     S = 0
                else
                     go to try_again;
```

11

Does the Implementation Work?

$S = 1$



$R_1\ R_2\ W_1\ W_2 \Longrightarrow$ ________________

12

Solutions

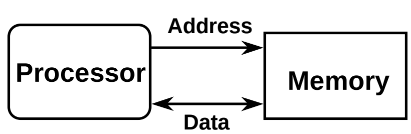
- Fundamental problem: Read-test-write of the semaphore must be an <u>atomic action</u>

- Solutions are:
  - Disable interrupts → Dangerous to give user Interrupt Enable Control!
  - Provide new memory operation
  - Get VERY clever (cf. Dekker, Dijsktra)

13

Atomic Test-and-Set

- New memory operation



- New transaction
  - Read
  - Write
  - Test-and-Set — atomically — return Memory[addr]; Memory[addr] ← 0

14

Using Atomic Test-and-Set

```
Test-and-set ( addr )
    return Memory[addr]
    Memory[addr] ← 0
```

```
signal ( S)
   S = S + 1 ;
```

```
wait ( S)
   try_again:  if  ( Test-and-Set( addr(S) ) == 0 ) then
                  goto try_again ;
```

15

Efficiency Issues

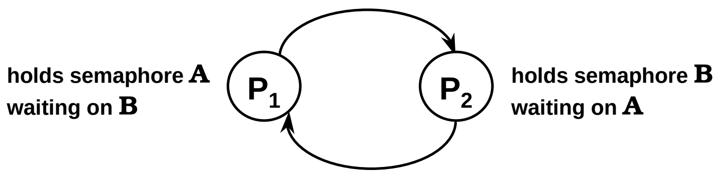
- Busy-waiting in the wait() procedure is undesirable in time-sharing systems
  – Busy-waiting process ties up the processor, but does not execute any useful instructions
- Say process Q has to perform signal(S) to allow waiting process P to run
  – More efficient to allow Q to run

```
wait ( S)
   try_again:  if  ( Test-and-Set( addr(S) ) == 0 ) then
                  {   yield() ;
                      goto try_again ;
                  }
```

16

Deadlock a.k.a. "The Deadly Embrace"

- **Indiscriminate use of semaphores can lead the system to be stuck in a state from which it never emerges**
  - **Directed cycle in interprocess dependencies**
  - **Process P waiting for Q which is waiting for P**



17

Deadlock Example

```
Transfer_Money( A, B, M ) {
      wait ( A.sem )
      wait ( B.sem )
      A.Balance  =  A.Balance  -  M
      B.Balance  =  B.Balance  +  M
      signal ( A.sem )
      signal ( B.sem )
}
```

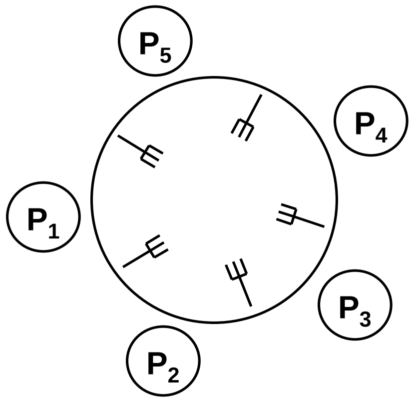
P_1 wishes to execute Transfer_Money(X, Y, 100.00)
P_2 wishes to execute Transfer_Money(Y, X, 10.00)

Deadlock occurs if P_1 is stuck on Y.sem
and P_2 is stuck on X.sem

18

6004 Spring 1998: L45

Dining Philosophers Problem



Philosophers' Algorithm

Take right fork
Take left fork
EAT
Replace both forks

Problem: Each philosopher may pick up right fork, and ____________________________

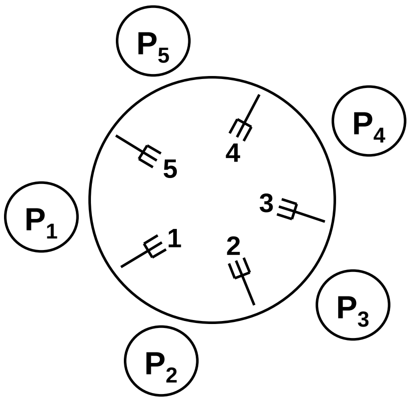
19

Deadlock Avoidance

- **Impose some discipline on the way in which resources, i.e., semaphores, are accumulated by a process, so deadlock becomes impossible**
- **Put all the resources in some numerical order and then require that any program always acquires resources in increasing numerical order**
  - **Assume program needs resources i and j, with i > j**
  - **Program will never try to acquire resource i if it is not already holding resource j**

20

Dining Computer Scientists



Computer Scientists' <u>Algorithm</u>

Take lower-numbered adjacent fork
Take higher-numbered adjacent fork
EAT
Replace both forks

CLAIM: Deadlock is impossible!

P_i holding fork i deadlock will never occur because P_2 will never hold fork 2 without holding fork 1 also.

21