

Sheets for MIEIC's SO

*based on teaching material supplied by
A. Tanenbaum for book:
Modern Operating Systems, ed...*

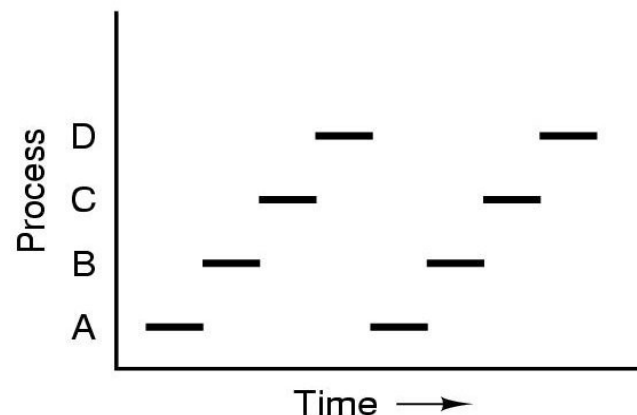
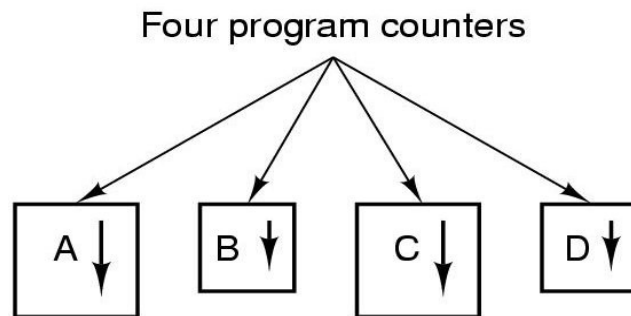
Chap 3: Coordination

Chapter 3

Coordination

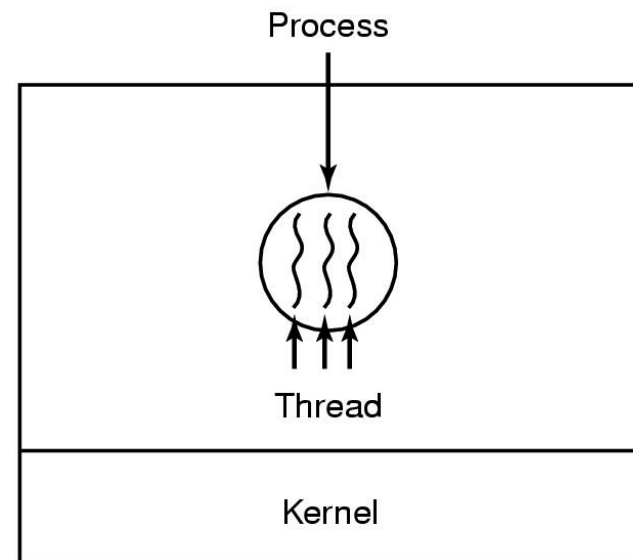
Interprocess communication (part 2)
Deadlocks

Concurrent Processes and Threads



Each program runs oblivious of when it will be “frozen” and another program will get the processor

The same can be said of the individual threads of each active process



Concurrency, Competition, Synchronization

Both processes and threads can compete to use available resources at the “same time”

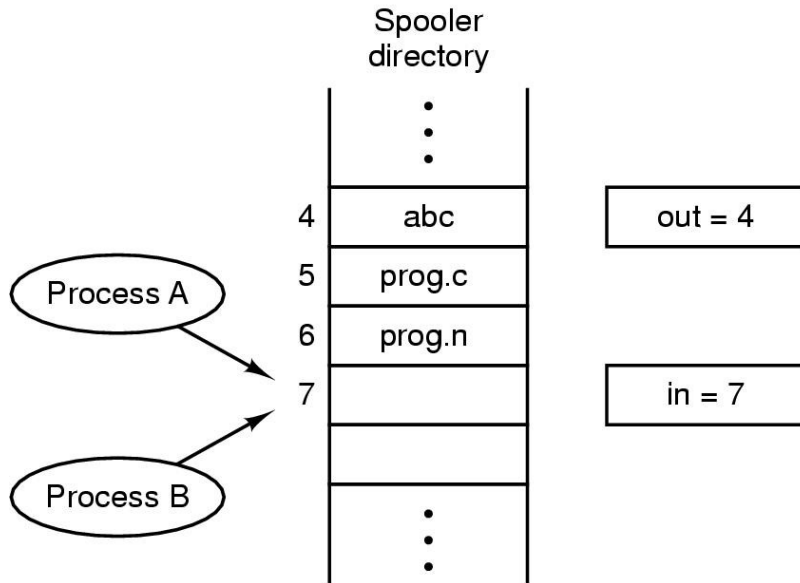
- threads do it more easily, because they share their process' address space, open files...

Competition leads to “race conditions”

- in which the result of the dispute is undefined

Some kind of arbitration (coordination, synchronization) is clearly needed!

Interprocess Communication: Race Conditions



Two processes want to
access shared memory
at same time

Spooler access w/o arbitration:

Proc.A: reads in (=7)

Sched: B's turn!

Proc.B: reads in (=7)

Proc.B: inserts docB in slot 7

Proc.B: writes 8 to in

Sched: A's turn!

Proc.A: inserts docA in slot 7
(overwriting docB!)

Proc.A: writes 8 to in

Sched: Spooler's turn!

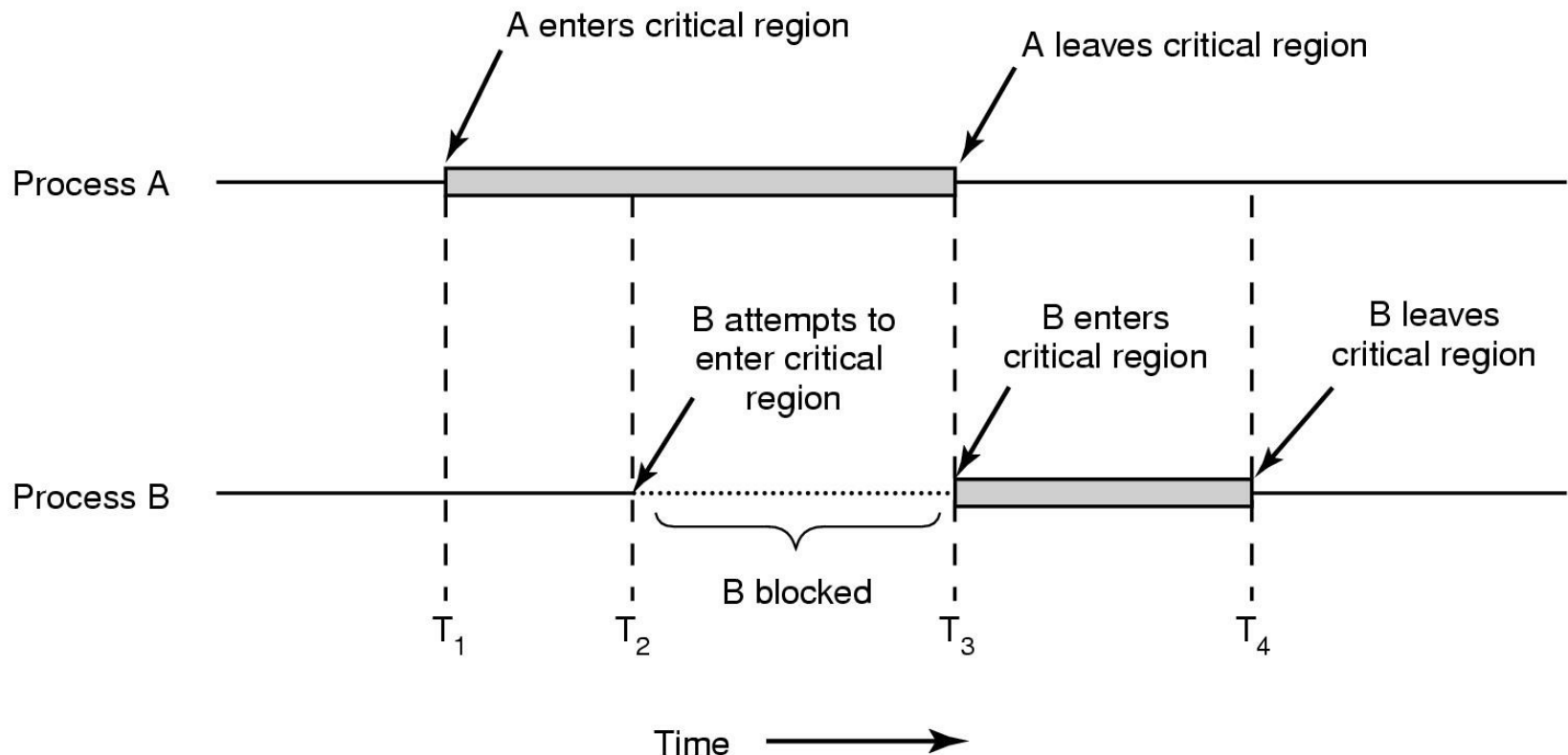
Spooler: prints docA
(and will never print docB!)

...

Critical Regions (1)

Part of program where shared resources are accessed!

Not more than one process/thread should execute it at the “same time” (*mutual exclusion*)!



Critical Regions (2): conditions

4 conditions for “good” *mutual exclusion*:

1. No two processes will be simultaneously in critical region
 - serialize access!
2. No assumptions will be made about speed or number of CPUs
 - use additional mechanisms to always make it work!
3. No process running outside the critical region will block another process
 - reservations are not allowed!
4. No process will wait forever to enter its critical region
 - starvation is not admissible!

How to achieve this Mutual Exclusion?!...

Critical Regions (3): synchronization of access

Processes should, in sequence:

1. wait until access to critical region is granted:

e.g. `enter_region()`;

2. go inside critical region:

e.g. `critical_region()`;

3. declare that they have left critical region

e.g. `leave_region()`;

```
...  
enter_region();    // wait...  
critical_region(); // access it  
leave_region();    // next!  
...
```


Mutual Exclusion (1): low-level help

Help from Hardware/Operating System:

- temporary disabling of interrupts, so scheduler will not run!
 - ok for kernel processes; not ok for user processes!
- *Test and Set Lock* instruction (w/ lock variable)
 - ok, but... *busy waiting*!

enter_region:

TSL REGISTER, LOCK

*shared
variable*

CMP REGISTER, #0

JNE enter_region

RET | return to caller; critical region entered

| copy lock to register and set lock to 1

| was lock zero?

| if it was non zero, lock was set, so loop

leave_region:

MOVE LOCK, #0

RET | return to caller

| store a 0 in lock

Mutual Exclusion (2): a software "solution"

A software “solution”:

- strict alternation :-(
– busy waiting!... :-(

Process 0:

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

Process 1:

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

Mutual Exclusion (3): another software solution

Peterson's software solution : be courteous!
(and *busy wait...*)

```
#define FALSE 0
#define TRUE 1
#define N      2    /* number of processes */

int turn;            /* whose turn is it? */
int interested[N];   /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;          /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = other;         /* set flag */
    while (turn == other && interested[other] == TRUE) /* null statement */ ;
}

critical_region();

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

can be generalized to 2+

critical_region();

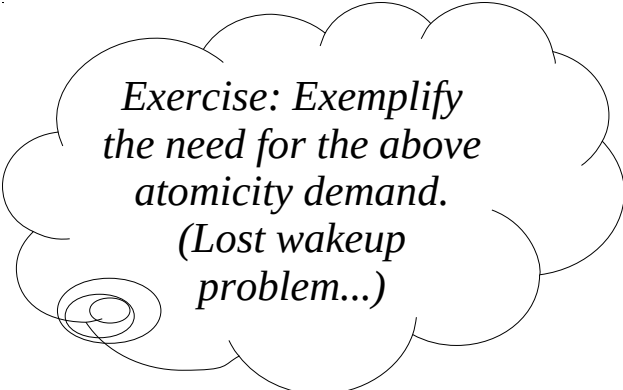
Mutual Exclusion *without* Busy Waiting (1)

Operating System “solution”? Sleep() + Wakeup()

```
region_id R;           // identifier of critical region
boolean r = 0;         // associated shared variable

enter_region(R) {
    if (r)              // this cannot be separated from
        Sleep(R);      // this (atomicity)! Otherwise...
    r = 1;
}

leave_region(R) {
    r = 0; Wakeup (R);
}
```



*Exercise: Exemplify
the need for the above
atomicity demand.
(Lost wakeup
problem...)*

Mutual Exclusion *without* Busy Waiting (2)

Operating System solution: Mutex

```
region_id R; // identifier of critical region
mutex M = UNLOCKED; // associated shared variable

enter_region(R) {
    Lock(M); // sleep if M was LOCKED;
              // otherwise proceed (and M goes LOCKED)
}

leave_region(R) {
    Unlock(M); // wake up someone or M goes UNLOCKED
}
```

To be seen:

mutex is a particular type of **Semaphore** : binary semaphore!

Mutual Exclusion *without* Busy Waiting (3)

Programming Language solution: Monitor

Only one process is allowed in monitor, i.e. executing any one of its procedures!

Java: **synchronized** methods!

While one thread is executing a synchronized method no other thread is allowed to execute synchronized methods for the same object.

```
monitor example
  integer i;
  condition c;

  procedure producer( );
  .
  .
  .
  end;

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

Mutual Exclusion *without* Busy Waiting (4)

(System) Library solution: Message Passing

– also for distributed systems! :-)

```
message M = "token";           // non-shared variable!  
  
some_process_gets_token(); // initialization  
  
enter_region(M) {  
    if (!has_token)  
        receive(*, M);    // blocking call  
}  
  
leave_region(M) {  
    send(a_process, M);  
}
```

What if token is “lost”?...

General Exclusion *without* Busy Waiting (1)

Operating System solution: Semaphores

```
semaphore S = N;  // shared variable; N: 0+

enter_region(S) {
    Down(S);  // decrement S; sleep if S would go < 0
              // otherwise proceed
}

leave_region(S) {
    Up(S);    // increment S;
              // wake up "someone" if S would not go > 0
}
```

Semaphore is a *kind of* counter of processes/threads:

- if > 0 : number allowed to be in same region
- [if would be < 0 : number waiting for entering the region]

if $S = 0..1$: binary semaphore or **mutex**

POSIX:

Down: `sem_wait()`

Up: `sem_post()`

General Exclusion *without* Busy Waiting (2)

Oper. Sys./Progr. Lang. solution: Condition Variables

```
condition C;          // shared variable
critical_region { // somehow, enter in exclusivity *
    ...
    if (cannot_proceed) wait (C);
        // atomicity is granted! Why?
        // while waiting, "someone" else may enter region
    ...
    if (wanna_leave) signal (C);
        // wake up "someone" and leave immediately
}
```

* For example, using a mutex!

POSIX:

pthread_cond_wait()
pthread_cond_signal()

Exercise*: identify race condition!

```
void fsin() { ... }  
int main() {  
    struct sigaction ss;  
    ... // set up ss with handler fsin();  
    sigaction(SIGUSR1, &ss, NULL);  
    int f = fork();  
    switch (f) {  
        case -1: perror("fork()"); exit(1);  
        case 0: pause(); printf("hello!"); break;  
        default: printf("World: "); kill(f, SIGUSR1);  
    }  
}
```



Sometimes, child process waits forever...
Where does the problem lie?

Deadlocks

What if all processes are waiting for one another?
(e.g. for entering a critical zone?)
→ a deadlock occurs!

- Mutual Exclusion can lead to deadlock.
- Synchronization in general can lead to deadlock too.
- Small coding errors can be fatal (on this respect):
 - race conditions always lurk...
 - signals can be lost...
 - processes can go sleeping before waking up another...

The problem has been studied for ages:
famous IPC problems (*next*)

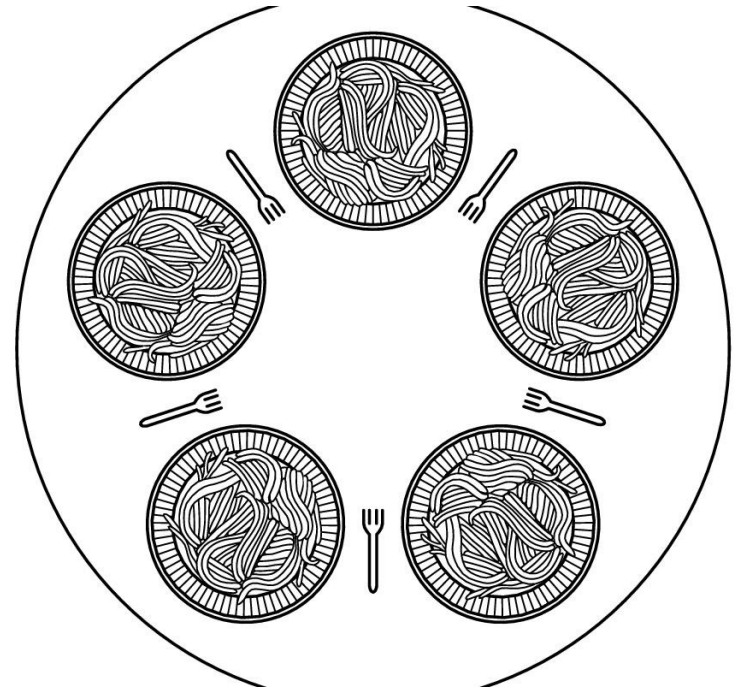


Inter Process Communication

Famous IPC problems (1)

Dining Philosophers

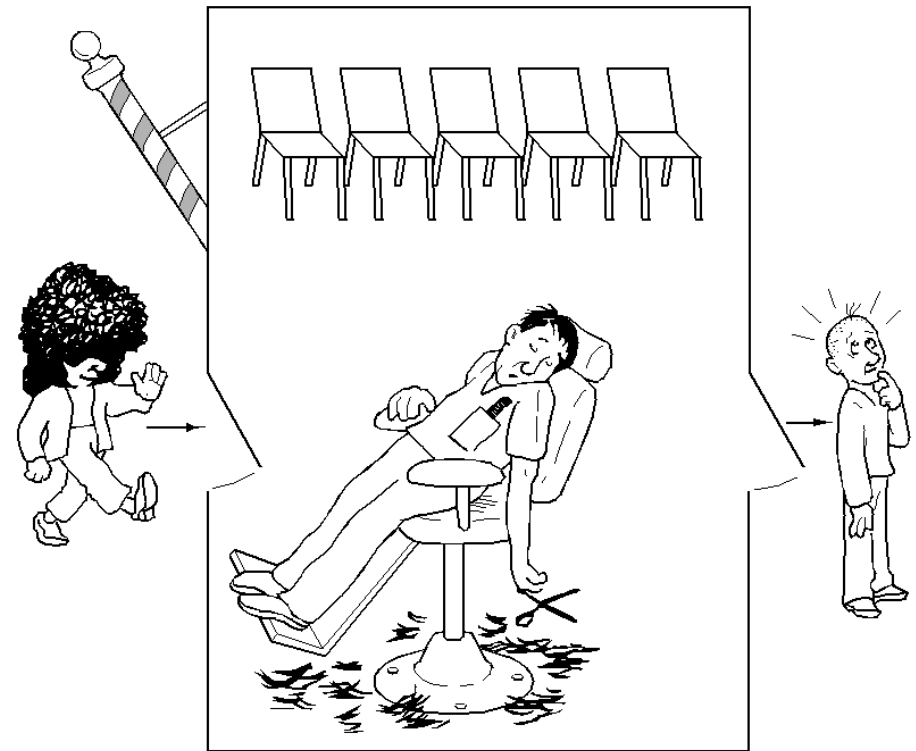
- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock?
 - (and starvation?)



Famous IPC problems (2)

Sleeping Barber

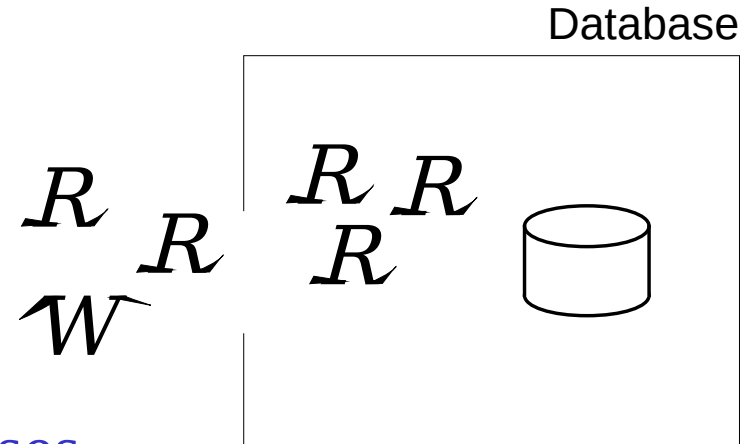
- Barber sleeps, waiting for customers
- Barber can serve one customer at a time
- Customer either
 - waits if there is an empty chair
- or
 - leaves if there are no empty chairs



FE3!

Famous IPC problems (3)

Readers and Writers



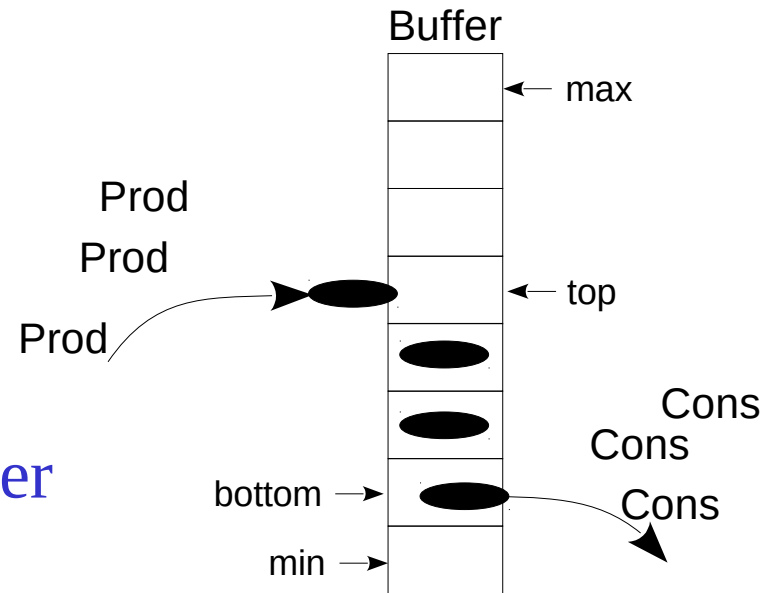
- A database record can be read by several simultaneous processes
- It can be written by a single process at a time
- How to:
 - prevent starvation (specially of writers)?
 - allow maximum throughput?



Famous IPC problems (4)

Bounded-Buffer (or Producer-Consumer)

- Producer puts goods into buffer (if it is not full)
- Consumer gets goods out of buffer (if it is not empty)
- How to synchronize their access to buffer?



FE3!

Deadlock (2)

A process holds resource A and requests resource B
at same time another process holds B and requests A

→ both are unable to proceed with their work!

Definition :

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

If blocked, none of the processes can run, release resources, wake up other process

If running, none of the processes can do useful work and progress towards their objectives (also, they waste system resources)

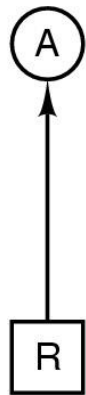
Mutual Exclusion can lead to deadlock...

Synchronization in general can lead to deadlock too...

→ it is a **concurrency** problem, in general!

Deadlock (3): modeling

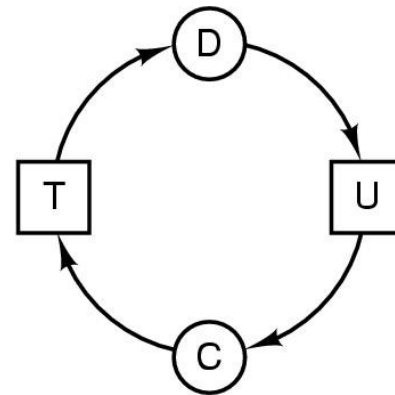
Modeled with directed graphs



(a)



(b)



(c)

- (a) resource R assigned to process A
- (b) process B is requesting/waiting for resource S
- (c) process C and D are in deadlock over resources T and U

Deadlock (4): pre-requisites

(excluding programming errors!)

- Mutual exclusion condition
 - each resource is assigned to 1 process or is available
- Hold and wait condition
 - process holding resources can request additional ones
- No preemption condition
 - previously granted resources cannot be forcibly taken away
- Circular wait condition
 - there is a circular chain of 2 or more processes, each waiting for a resource held by next member of the chain

Deadlock (5): dealing with it

Strategies:

- just ignore the problem altogether
 - ok, if it is rare or does not “hurt” much when happens
- detection and recovery
 - monitor system and terminate (some of) deadlocked processes
- prevention
 - negate one of the four necessary conditions
- avoidance
 - do careful resource allocation