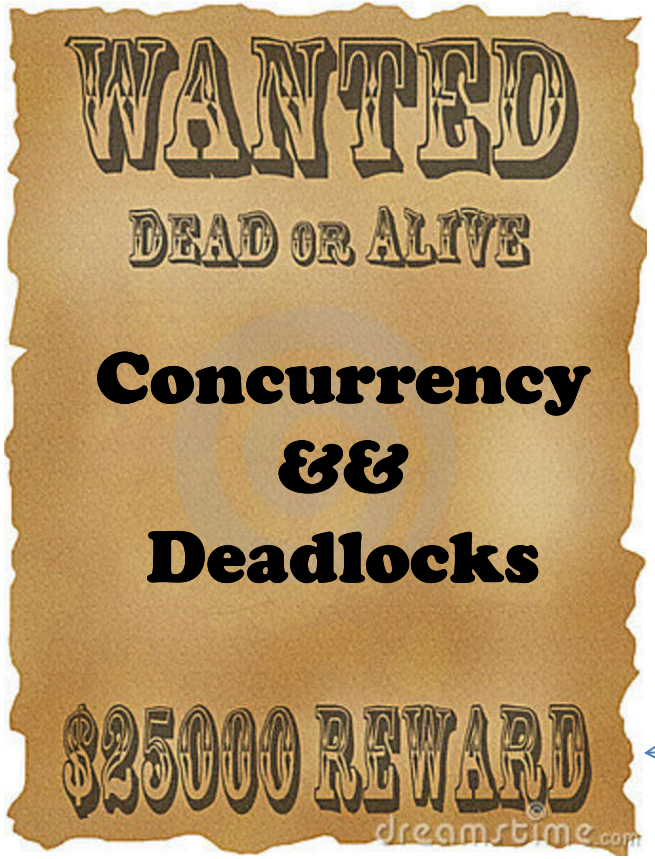
CSCI-GA.2250-001

Operating Systems

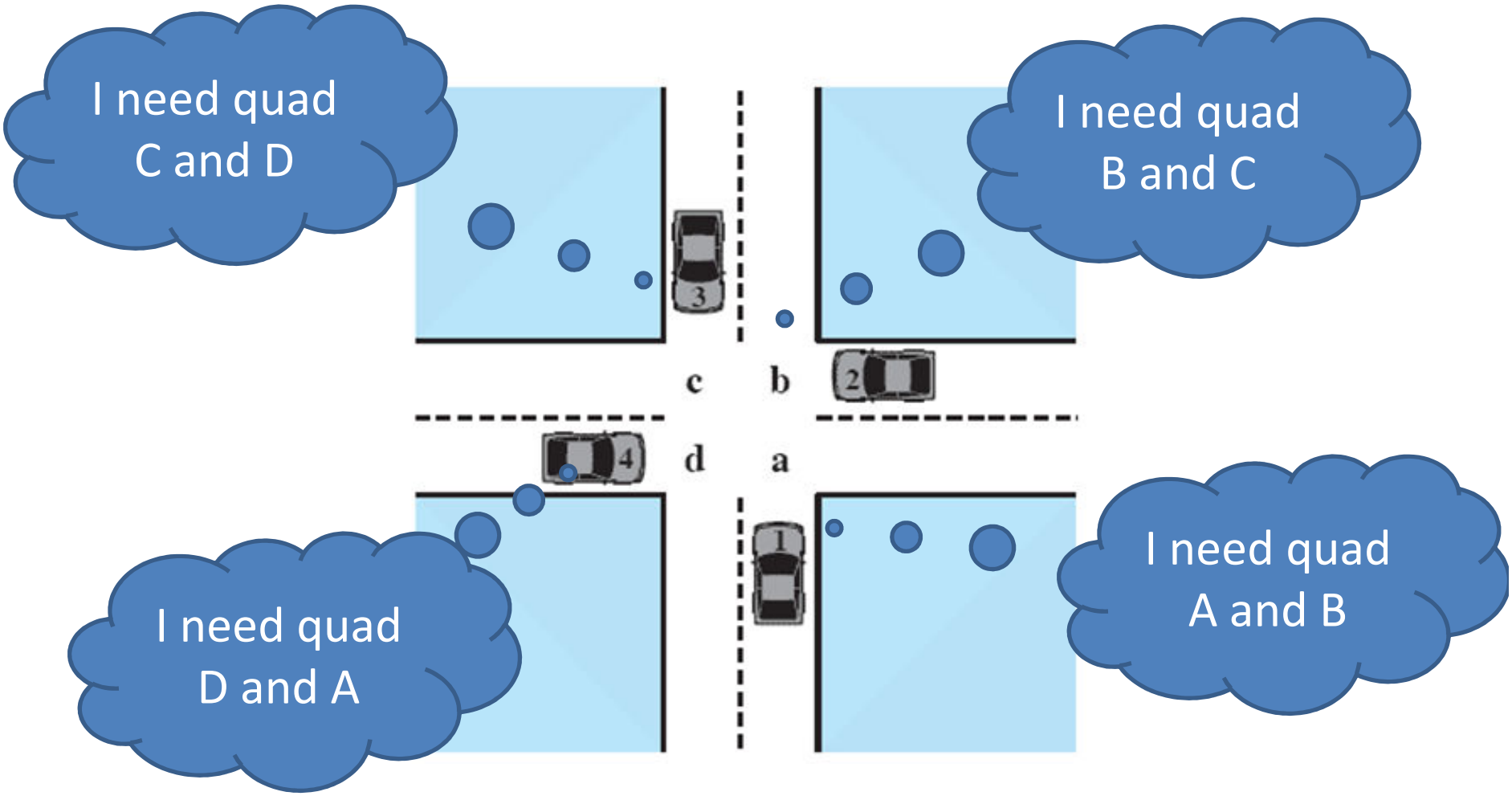


For illustration purpose only

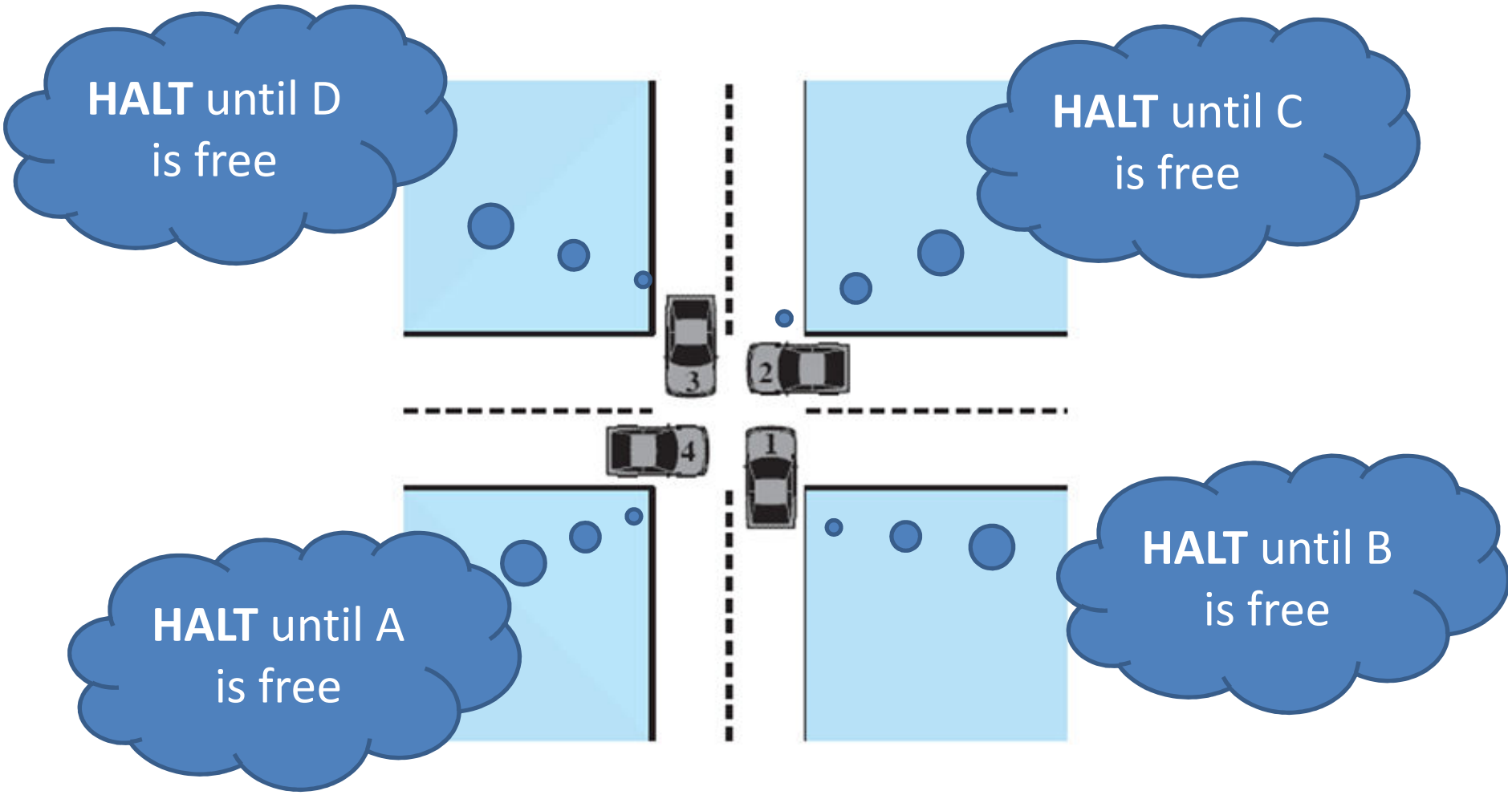
Hubertus Franke
frankeh@cs.nyu.edu


BANCO REAL
11455

Potential Deadlock



Actual Deadlock



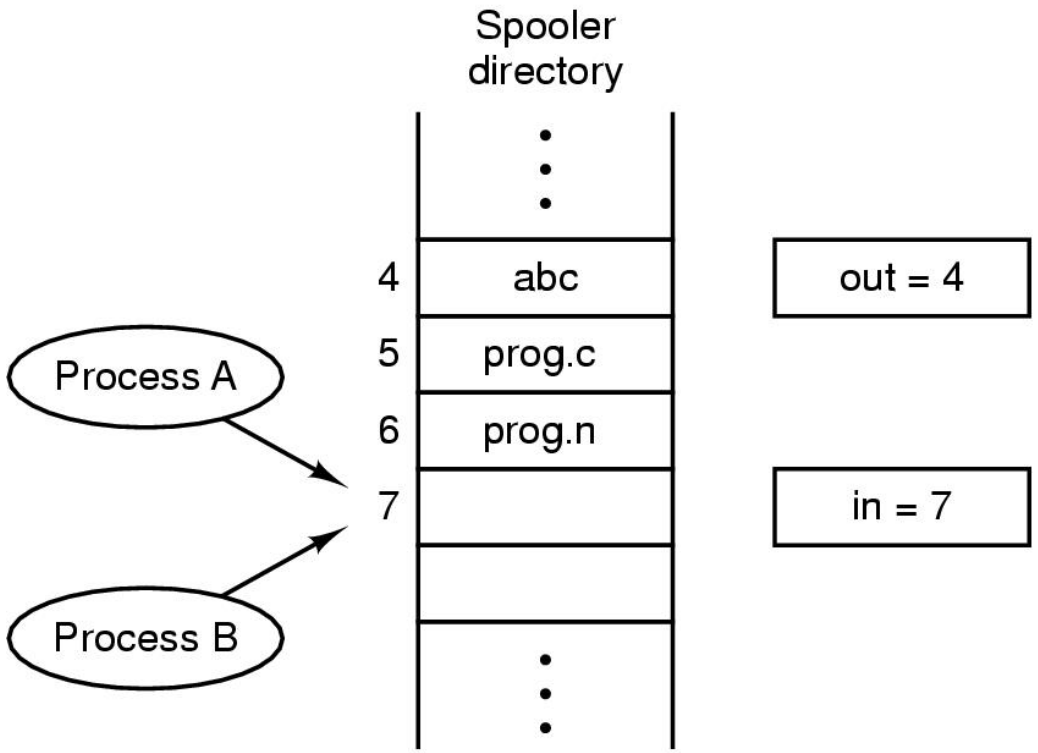
Inter Process Communication (IPC)

- Processes (or threads) need to work together (or at the very least share resources).

  Issues:
  - **Send information**
  - **Mitigate contentions over resources**
  - **Synchronize dependencies**

Inter-process Communication

Race Conditions: result depends on exact order of processes running



Two processes want to access shared memory at same time:
Read is typically not an issue but write or conditional execution IS

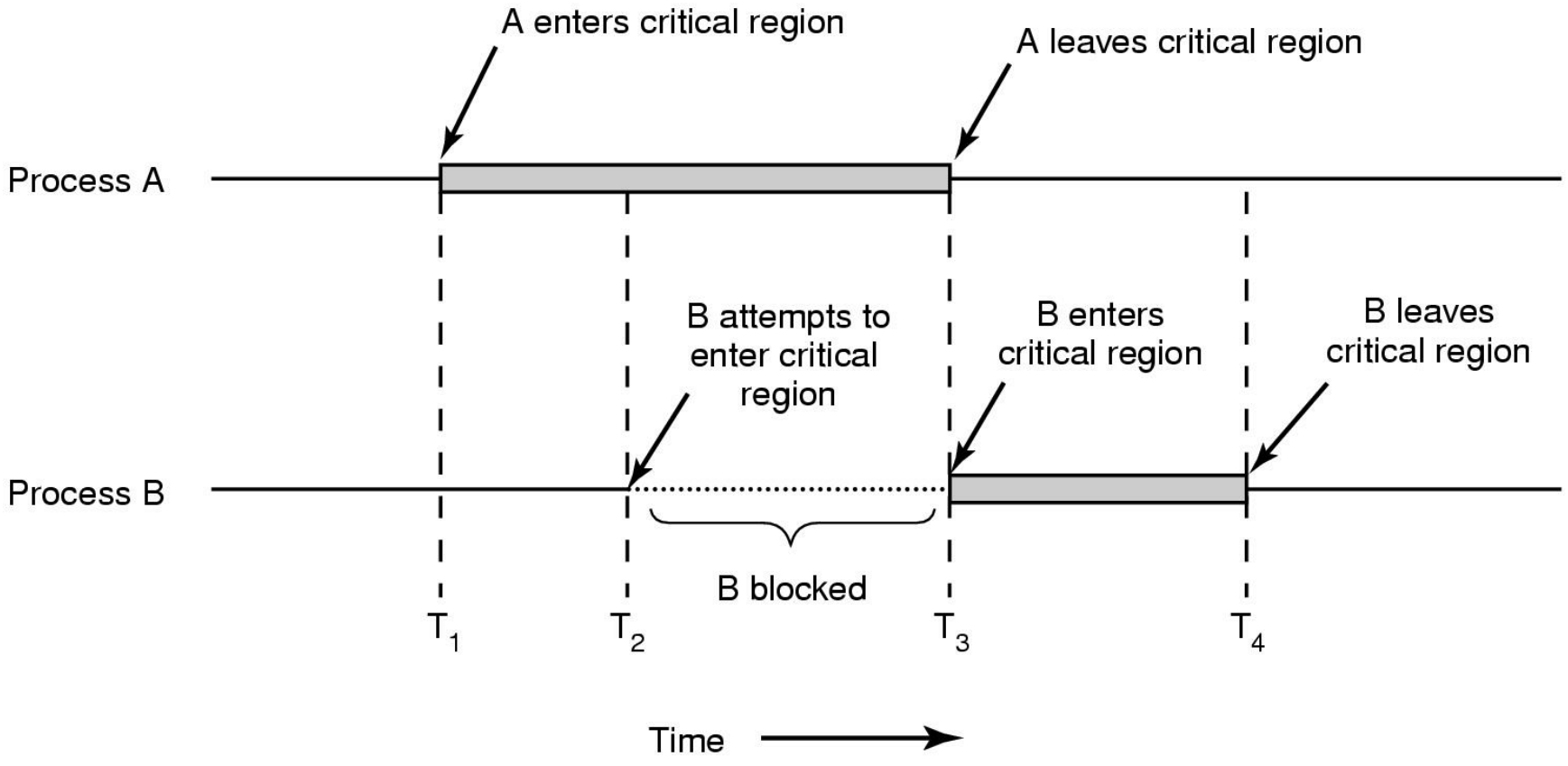
Critical Regions

Mutual Exclusion: Only one process at a time can access any shared variables, memory, or resources.

Four conditions to prevent errors:

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Regions



Mutual exclusion using critical regions

Mutual Exclusion with Busy Waiting

Simplest solution?

- How about disabling interrupt?

User program has too much privilege.

Lock variable?

- If the value of the lock variable is 0 then it sets it to "1" and enters. Other processes have to wait.
- What's the problem here?

Mutual Exclusion with Busy Waiting

```
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)

Proposed solution to critical region problem
(a) Process 0. (b) Process 1.

Mutual Exclusion with Busy Waiting

```
#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 = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

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

Peterson's solution for achieving mutual exclusion

The TSL (Test and Set Lock) Instruction

- With a "little help" from Hardware

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was nonzero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

The XCHG Instruction

```
enter_region:
        MOVE REGISTER,#1            | put a 1 in the register
        XCHG REGISTER,LOCK          | swap the contents of the register and lock variable
        CMP REGISTER,#0             | was lock zero?
        JNE enter_region            | if it was non zero, lock was set, so loop
        RET                         | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                | store a 0 in lock
        RET                         | return to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction.

Solutions so far

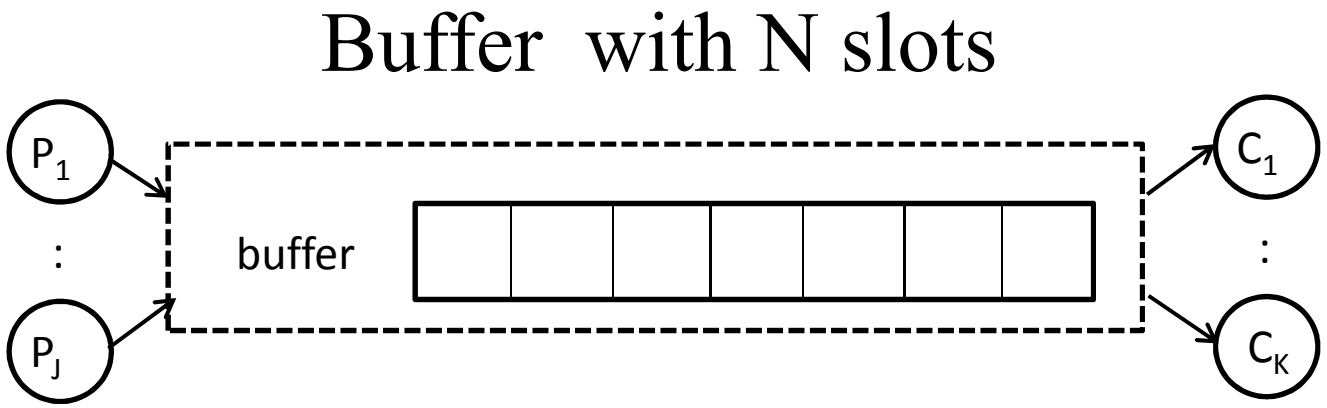
- Both TSL and Peterson's solutions are correct
  - But they have busy-waiting
  - Waste CPU time
  - Priority Inversion problem

Priority Inversion Problem

- Higher priority process can be prevented from entering a CS because the lock variable is dependent on a lower priority process.
  - Priority P1 < Priority P2
  - P1 is in its CS but P1 is never scheduled. Since P2 is busy in busy-waiting using the CPU cycles

Concurrency

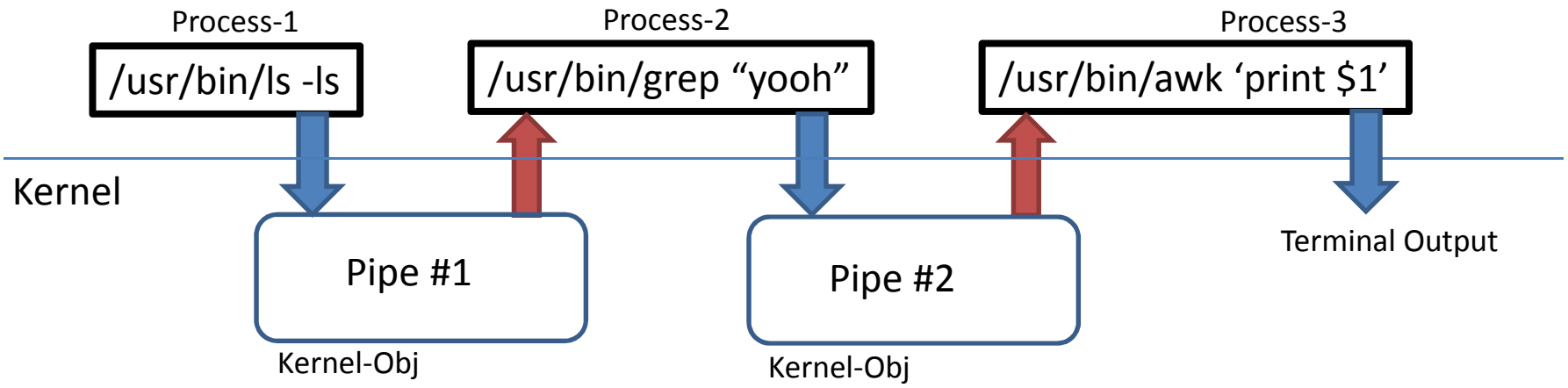
Producer-consumer problem



- Producer:
  - How do I know a slot is free ?
  - How do I know a slot became free ?
- Consumer:
  - How do I know nothing is available?
  - How do I know something became available ?

How can mutual exclusion solutions be used in every day OSs?

- Example:
  - UNIX: ls –ls | grep "yooh" | awk 'print $1'



- Responsibility of a Pipe
  - Provide Buffer to store data from stdout of Producer and release it to stdin of Consumer
  - Block Producer when the buffer is full (because consumer has not consumed data)
  - Block Consumer if no data in buffer when the consumer wants to read(stdin)
  - Unblock Producer when buffer space becomes free
  - Unblock Consumer when buffer data becomes available

Pipes

- Pipes not just for stdin and stdout
- Pipes can be created by applications
- All kind of usages for pipes.

- PipeBuffer typically has 16 write slots
- 4KB guaranteed to be atomic

```
frankeh@GCD:~$ cat /proc/sys/fs/pipe-max-size
1048576
```

```
NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Obtain O_* constant definitions
*/
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe()  creates a pipe, a unidirectional data channel that can be
       used for interprocess communication.  The array pipefd is used to
       return  two  file  descriptors referring to the ends of the pipe.
       pipefd[0] refers to the read end of the pipe.   pipefd[1]  refers
       to  the  write end of the pipe.  Data written to the write end of
       the pipe is buffered by the kernel until it is read from the read
       end of the pipe.  For further details, see pipe(7).
```

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     [redacted] {                               /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     [redacted] {                               /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer);  /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

```
#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) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (true) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Called after pre-emption but before sleep in the consumer

Pre-emption!!!
Count == 0

Fatal Race Condition

1. Let count = 1

2. Consumer begins loop, decrements count == 0

3. Consumer returns to loop beginning and executes:
   if(count == 0), then
   pre-emption happens.

4. Producer gets to run, executes
   count = count + 1;
   if(count == 1) and calls wakeup(consumer)

5.Pre-emption Consumer calls sleep(consumer)

Semaphore Data Structure

```
class Semaphore
{
        int  value;     /* counter */
        List queue;     /* list of processes sleeping
                           in this semaphore */
        void P();       /* down(), wait() */
        void V();       /* up(), signal () */
}
```

Dijkstra 1965

Semaphore implementations

```
P()
{
        value = value –1;
        if value < 0
        {
                add the calling process to this
          semaphore's list;
              Sleep();
        }
}
```

Semaphore implementations

```
V()
{
        value = value +1;
        if value <= 0
        {
            remove a process from this semaphore's list;
            wake up the sleeping process;
        }
}
```

Semaphore Solution

Q) How do P() and V() avoid the race condition?

A) P() and V() are atomic.

- First line of P() & V() can disable interrupts.
- Last line of P() & V() re-enables interrupts.

Two kinds of semaphores

- Counting semaphores: for synchronization problems. value initialized to larger than 1
- Mutex semaphores (or binary semaphores): for mutual exclusion problems: value initialized to 1

Semaphore Solution To the Producer Consumer Problem

3 semaphores required

```
#define N 100                    /* number of slots in the buffer */
typedef int semaphore;           /* semaphores are a special kind of int */
semaphore mutex = 1;             /* controls access to critical region */
semaphore empty = N;             /* counts empty buffer slots */
semaphore full = 0;              /* counts full buffer slots */
```

In addition you need a write-index and a read-index
widx = (widx+1) % N ; ridx = (ridx+1) % N ;

```
void producer(void)
{
     int item;

     {
          item = produce_item( );
          down(&empty);
          down(&mutex);
          insert_item(item);
          up(&mutex);
          up(&full);
     }
}
```

```
void consumer(void)
{
     int item;

     {
          down(&full);
          down(&mutex);
          item = remove_item( );
          up(&mutex);
          up(&empty);
          consume_item(item);
     }
}
```

Mutex Implementation

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure 2-29. Implementation of mutex lock and mutex unlock.

Mutexes in Pthreads

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

Figure 2-30. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-31. Some of the Pthreads calls relating to condition variables.

Mutexes in Pthreads

```
#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);
}

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);
}
```

. . .

Figure 2-32. Using threads to solve the producer-consumer problem.

Problems with semaphore?

- It is difficult to write semaphore code
- One has to be careful in code construction
- If a thread dies and it holds a mutex semaphore, the implicit token is lost

Deadlock-free code

```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

Code with potential deadlock

```
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_2);
                down(&resource_1);
                use_both_resources( );
                up(&resource_1);
                up(&resource_2);
        }
```

Some Advice for locks

- Always release in reverse order as required
- Always acquire multiple locks in the same order
- Example: Multi Queue CPU scheduler where load balancing is required.
  (blackboard drawing)

Monitors

- Hoare and Brinch Hansen proposed a higher-level synchronization primitive: *monitor*

  - Only **ONE** thread allowed inside the Monitor
  - **Compiler** achieves Mutual Exclusion
  - Monitor is a **programming language** construct like a class or a for-loop.
  - We still need a way to synchronize on events!
    - Condition variables: wait & signal
    - Not counters: signaling with no one waiting → event lost
    - Waiting on signal releases the monitor and wakeup reacquires it.

Monitors

- What to do when a thread **A** in a monitor signals a thread **B** sleeping on a condition variable?
  - Ensure that signal is the last command in the monitor?
  - Block **A** while **B** runs?
  - Let **A** finish then run **B**?

Monitor Example

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

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

Figure 2-33. A monitor.

Monitors

```
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;
```

Figure 2-34. An outline of the producer-consumer problem with monitors.

Producer-Consumer Problem with Message Passing

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

  . . .
```

Figure 2-36. The producer-consumer problem with N messages.

Producer-Consumer Problem with Message Passing

```
void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);                  /* get message containing item */
                item = extract_item(&m);                /* extract item from message */
                send(producer, &m);                     /* send back empty reply */
                consume_item(item);                     /* do something with the item */
        }
}
```

Figure 2-36. The producer-consumer problem with N messages.

Message Passing

- No shared memory on distributed systems…

- Instead we can send LAN messages.

```
send(destination, &message);
receive(destination, &message);
```

Message Passing Issues

- Guard against lost messages (acknowledgement)

- Authentication (guard against imposters)

- Addressing :
  - To processes
  - Via Mailbox (place to buffer messages)
  - Send to a full mailbox means block
  - Receive from an empty mailbox means block

Message Passing Issues

- What about buffer-less messages:
  - Send and Receive wait (block) for each other to be ready to talk: rendezvous

Barriers

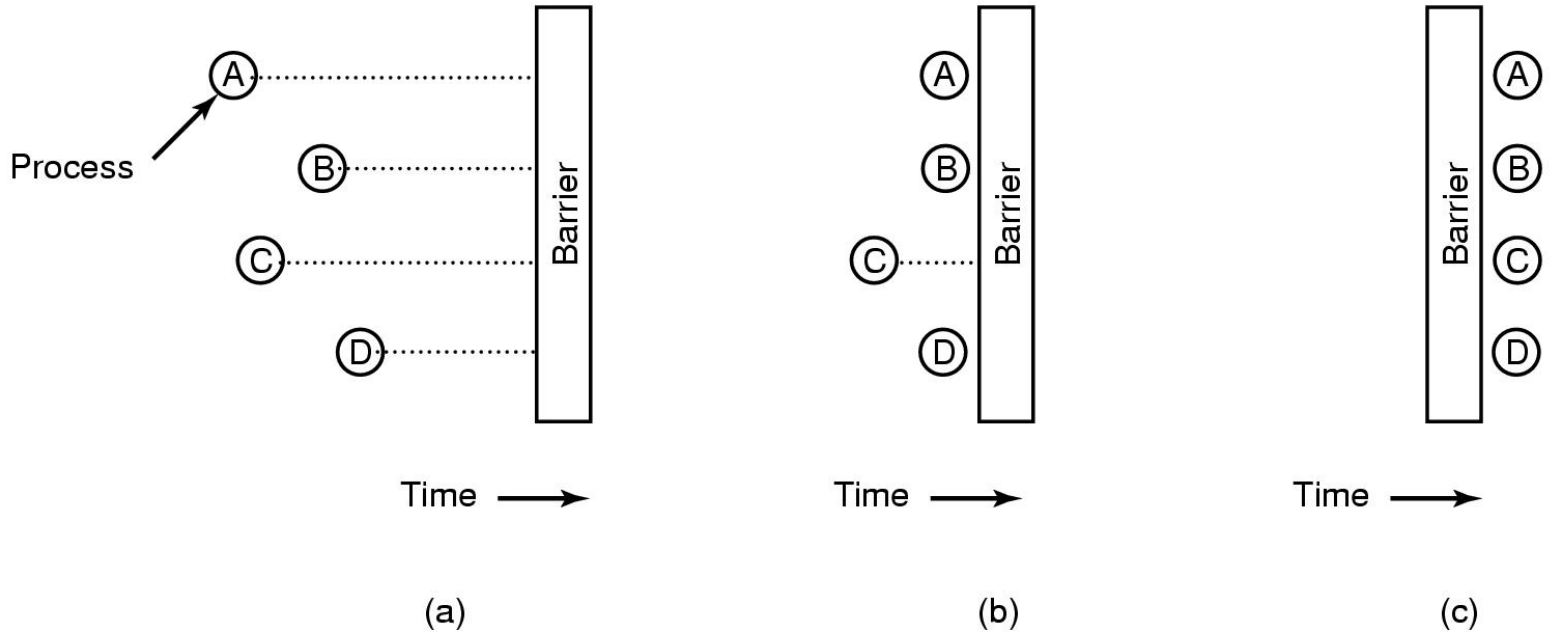


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Deadlock vs Starvation

- Deadlock – Process(es) waiting on events (resources) that will never happen:
  - Can be system wide or just one process
- Starvation – Process(es) waiting for its turn but never comes:
  - Process could move forward, the resource or event might be come available but this process does not get access.
  - Printing policy prints smallest file available. 1 process shows up with HUGE file, will not get to run if steady stream of files.

Deadlocks

Occur among processes who need to acquire resources in order to progress

Resources

- Anything that must be acquired, used, and released over the course of time.
- Hardware or software resources
- Preemptable and Nonpreembtable resources:
  - Preemptable: can be taken away from the process with no ill-effect
  - Nonpreemptable: cannot be taken away from the process without causing the computation to fail

Resource Categories

Reusable

- can be safely used by only one process at a time and is not depleted by that use
  - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, messages, and information
  - in I/O buffers

So …

- 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.
- Assumptions
  - If a process is denied a resource, it is put to sleep
  - Only single-thread processes
  - No interrupts possible to wake up a blocked process

Conditions for Resource Deadlocks

1. Each resource is either currently assigned to exactly one process or is available.
2. Processes currently holding resources that were granted earlier can request new resources.
3. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

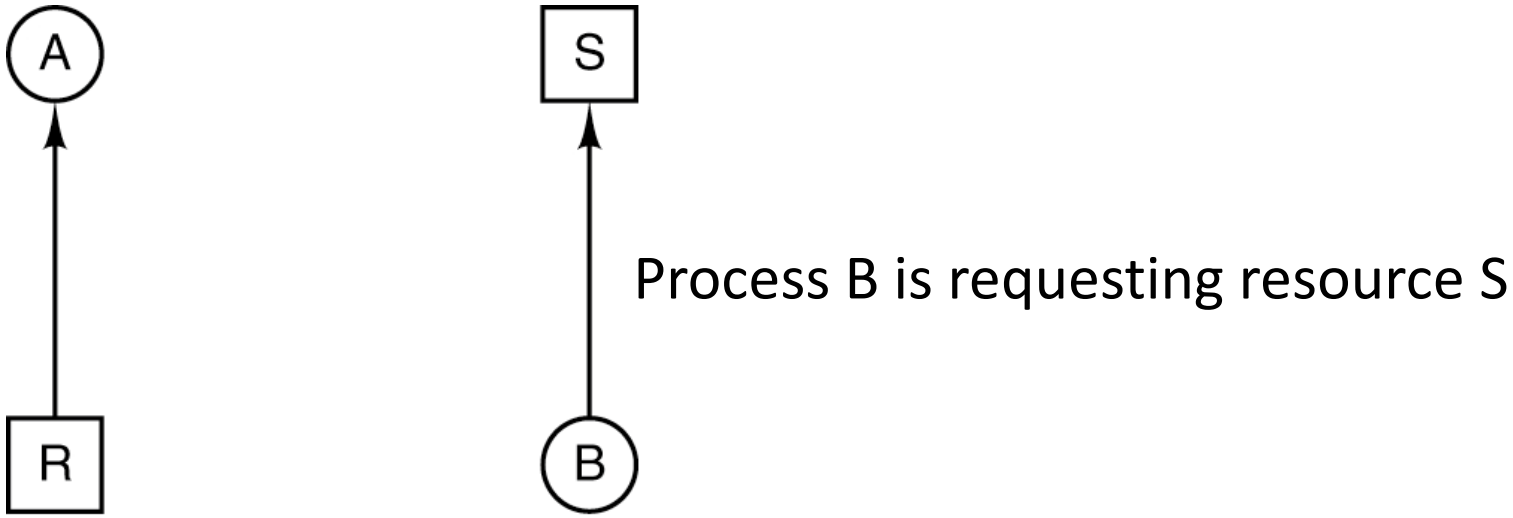
Resource Allocation Graph

Process

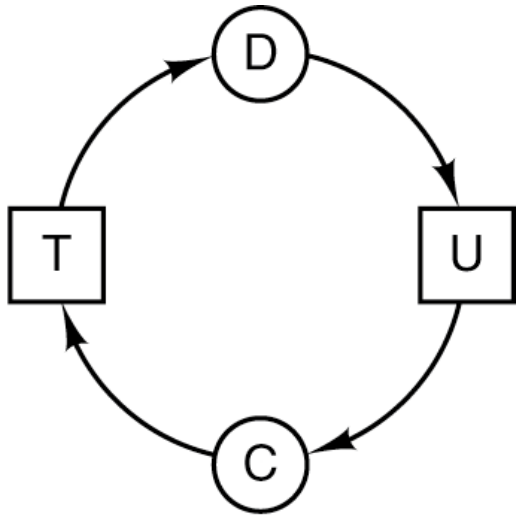
Process A is holding resource R

Resource

Resource Allocation Graph



Resource Allocation Graph



Deadlock!

A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

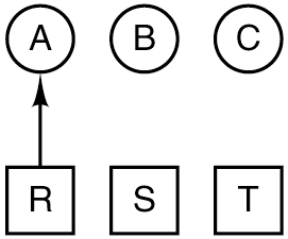
C

Request T
Request R
Release T
Release R

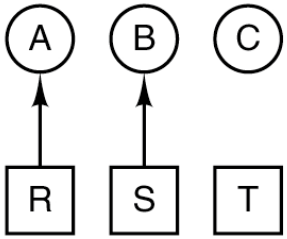
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

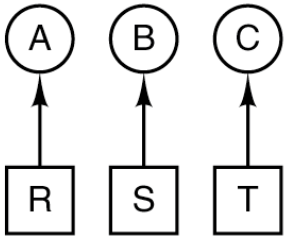
(d)



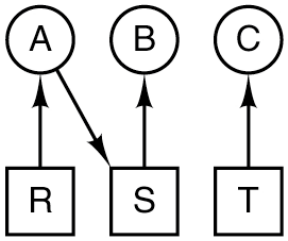
(e)



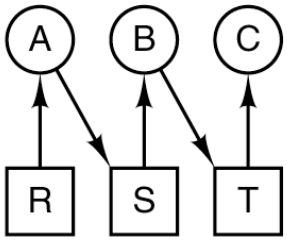
(f)



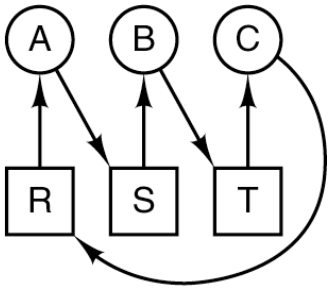
(g)



(h)



(i)



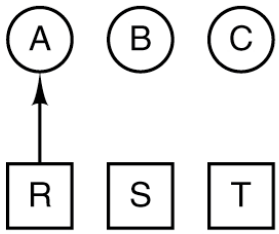
(j)

A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

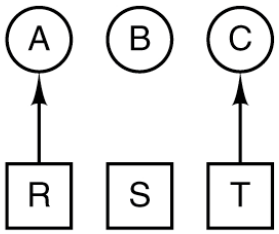
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

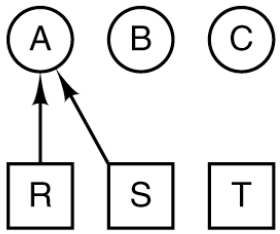
(k)



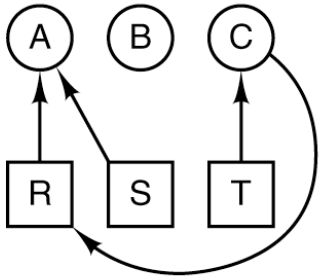
(l)



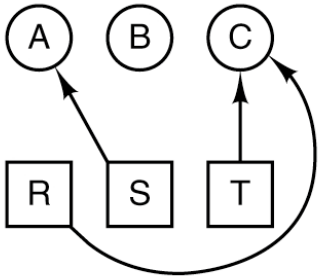
(m)



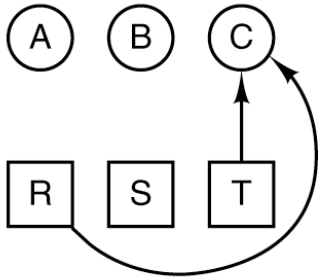
(n)



(o)



(p)



(q)

How to Deal with Deadlocks

1. Just ignore the problem!
2. Let deadlocks occur, detect them, and take action
3. Dynamic avoidance by careful resource allocation
4. Prevention, by structurally negating one of the four required conditions.

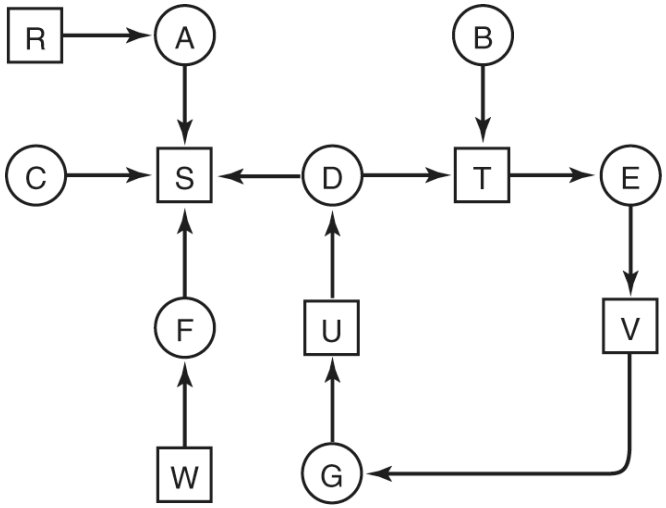
The Ostrich Algorithm

Deadlock Detection and Recovery

- The system does not attempt to prevent deadlocks.
- It tries to detect it when it happens.
- Then it takes some actions to recover
- Several issues here:
  - Deadlock detection with one resource of each type
  - Deadlock detection with multiple resources of each type
  - Recovery from deadlock

Deadlock Detection: One Resource of Each Type

- Construct a resource graph
- If it contains one or more cycles, a deadlock exists



Formal Algorithm to Detect Cycles in the Allocation Graph

For Each node N in the graph do:

1. Initialize L to empty list and designate all arcs as unmarked
2. Add the current node to end of L. If the node appears in L twice then we have a cycle and the algorithm terminates
3. From the given node pick any unmarked outgoing arc. If none is available go to 5.
4. Pick an outgoing arc at random and mark it. Then follow it to the new current node and go to 2.
5. If the node is the initial node then no cycles and the algorithm terminates. Otherwise, we are in dead end. Remove that node and go back to the previous one. Go to 2.

When to Check for Deadlocks?

- Check every time a resource request is made
- Check every k minutes
- When CPU utilization has dropped below a threshold

Recovery from Deadlock

- We have detected a deadlock … What next?
- We have some options:
  - Recovery through preemption
  - Recovery through rollback
  - Recovery through killing processes

Recovery from Deadlock: Through Preemption

- Temporary take a resource away from its owner and give it to another process
- Manual intervention may be required (e.g. in case of printer)
- Highly dependent on the nature of the resource.
- Recovering this way is frequently impossible.

Recovery from Deadlock: Through Rollback

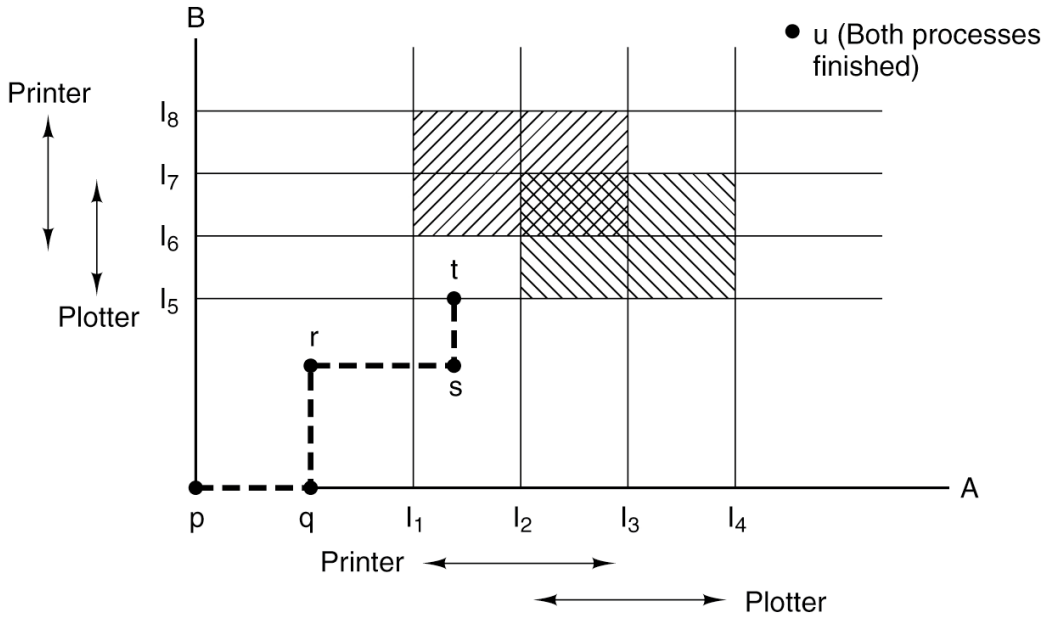
- Have processes checkpointed periodically
- Checkpoint of a process: its state is written to a file so that it can be restarted later
- In case of deadlock, a process that owns a needed resource is rolled back to the point before it acquired that resource

Recovery from Deadlock: Through Killing Processes

- Kill a process in the cycle.
- Can be repeated (i.e. kill other processes) till deadlock is resolved
- The victim can also be a process NOT in the cycle

Deadlock Avoidance

- In most systems, resources are requested one at a time.
- Resource is granted only if it is safe to do so



Safe and Unsafe States

- A state is said to be safe if there is one scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
- An unsafe state is NOT a deadlock state

Safe and Unsafe States

Maximum the process will need (e.g. A will need 6 more).

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

Assume a total of 10 instances of the resources available

This state is safe because there exists a sequence of allocations that allows all processes to complete.

Safe and Unsafe States

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

How about this state?

The difference between a safe and unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

The Banker's Algorithm

- Dijkstra 1965
- Checks if granting the request leads to an unsafe state
- If it does, the request is denied.

The Banker's Algorithm: The Main Idea

- The algorithm checks to see if it has enough resources to satisfy some customers
- If so, the process closest to the limit is assumed to be done and resources are back, and so on.
- If all loans (resources) can eventually be repaid, the state is safe.

The Banker's Algorithm: Example (single resource type)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

Safe

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

Safe

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Unsafe

The Banker's Algorithm: Example (multiple resources)

Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

The Banker's Algorithm

- Very nice theoretically
- Practically useless!
  - Processes rarely know in advance what their maximum resource needs will be.
  - The number of processes is not fixed.
  - Resources can suddenly vanish.

Deadlock Prevention

- Deadlock avoidance is essentially impossible.
- If we can ensure that at least one of the four conditions of the deadlock is never satisfied, then deadlocks will be structurally impossible.

Deadlock Prevention: Attacking the Mutual Exclusion

- Can be done for some resources (e.g the printer) but not all.
- Spooling
- Words of wisdom:
  - Avoid assigning a resource when that is not absolutely necessary.
  - Try to make sure that as few processes as possible may actually claim the resource

Deadlock Prevention:
Attacking the Hold and Wait Condition

- Prevent processes holding resources from waiting for more resources
- This requires all processes to request all their resources before starting execution
- A different strategy: require a process requesting a resource to first temporarily release all the resources it currently holds. Then tries to get everything it needs all at once

Deadlock Prevention:
Attacking No Preemption Condition

- Virtualizing some resources can be a good strategy (e.g. virtualizing a printer)
- Not all resources can be virtualized (e.g. records in a database)

Deadlock Prevention: The circular Wait Condition

- Method 1: Have a rule saying that a process is entitled only to a single resource at a moment.
- Method 2:
  - Provide a global numbering of all resources.
  - A process can request resources whenever they want to, but all requests must be done in numerical order
  - With this rule, resource allocation graph can never have cycles.

Deadlock Prevention: Summary

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Conclusions

- Deadlocks can occur on hardware/software resources
- OS need to be able to:
  - Detect deadlocks
  - Deal with them when detected
  - Try to avoid them if possible