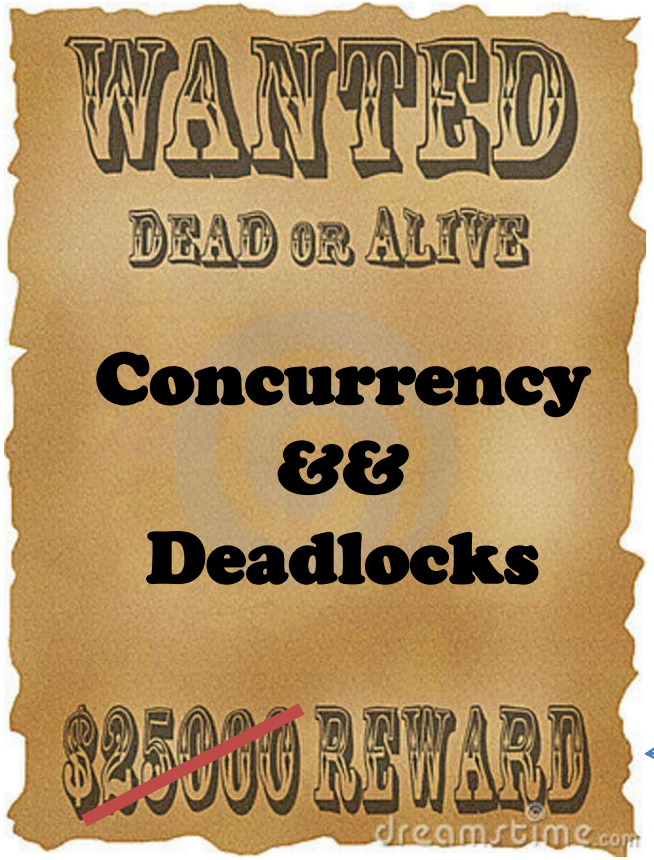
CSCI-GA.2250-001

Operating Systems



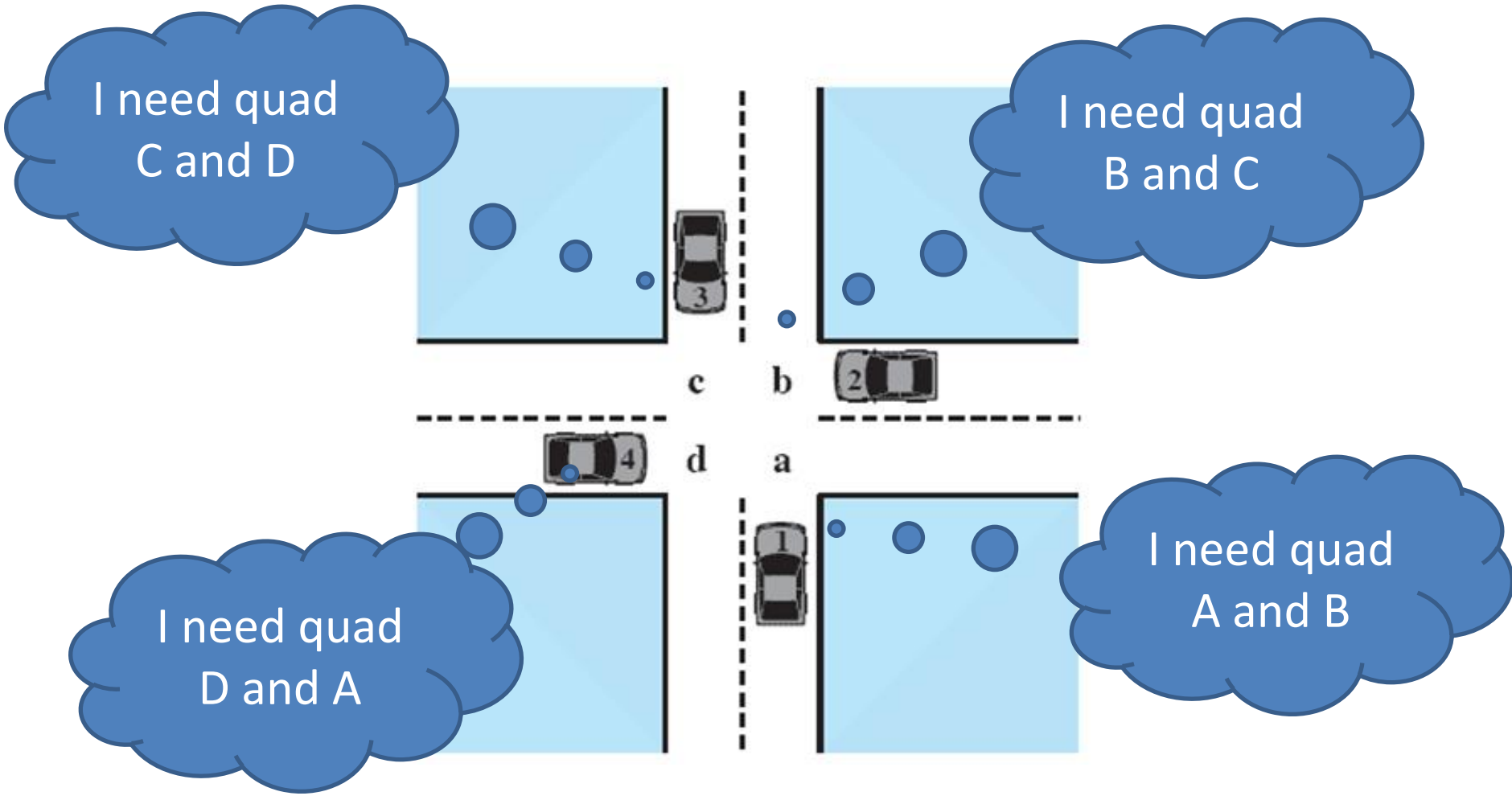
$30000
(inflation adjustment)

For illustration purpose only

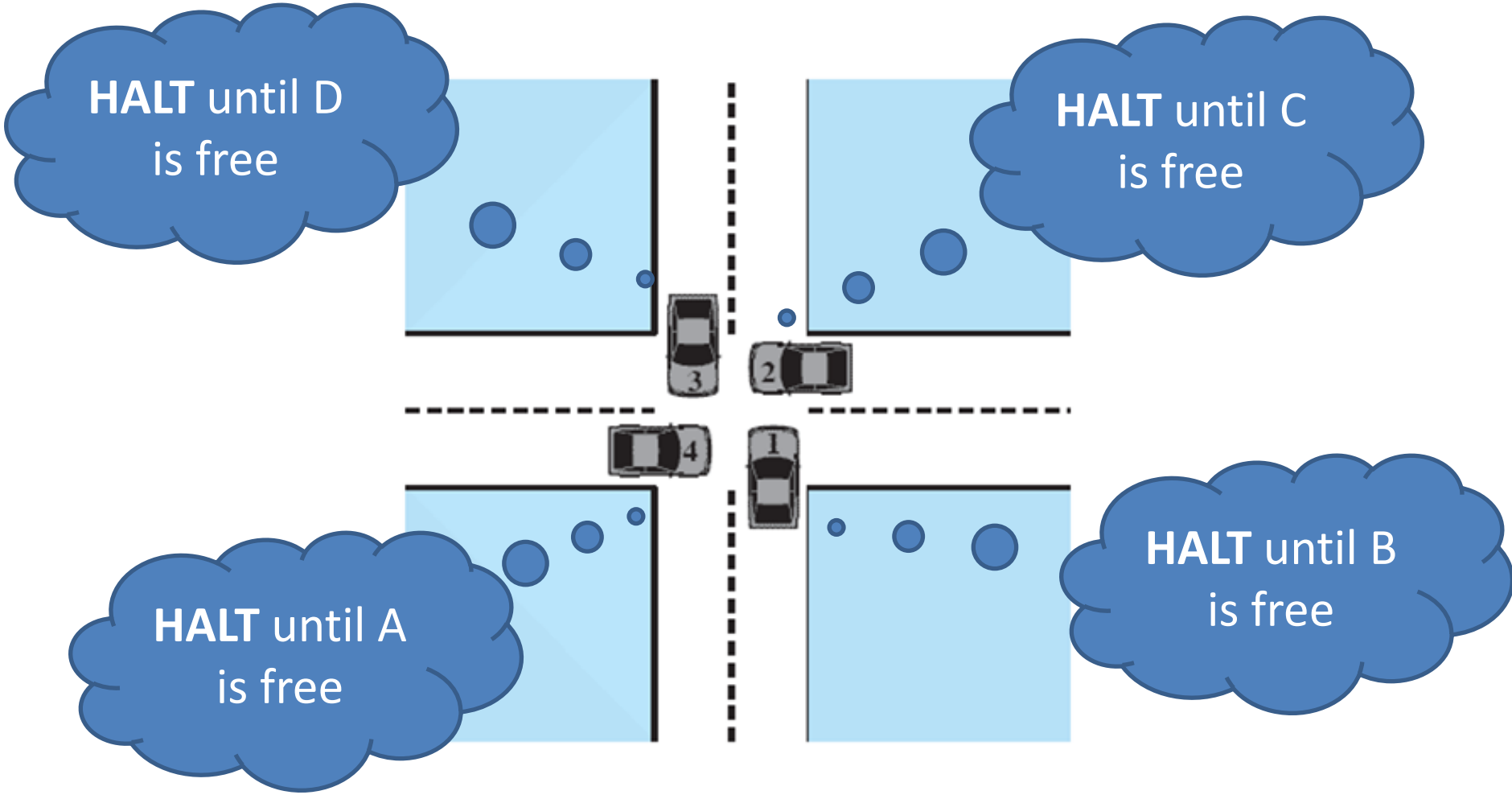
Hubertus Franke
frankeh@cs.nyu.edu


1455

Potential Deadlock



Actual Deadlock



Reminder

- Concurrency is having multiple contexts of execution not necessarily running at the exact same time.
- Parallelism is having multiple contexts of execution running at the exact same time.

Inter Process Communication (IPC)

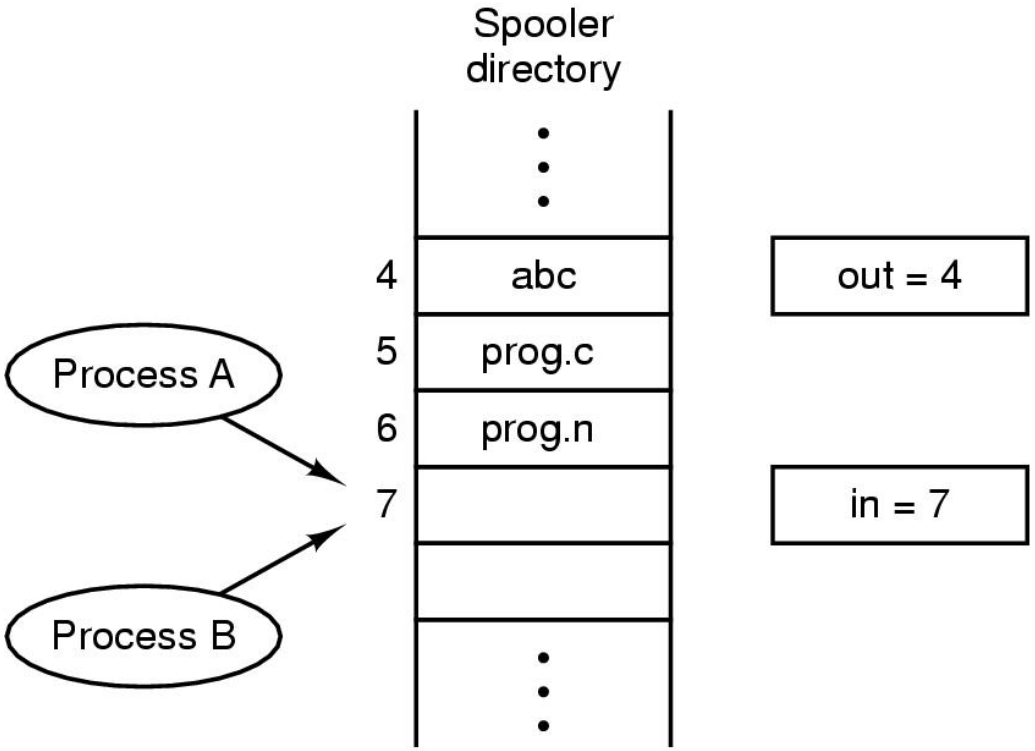
- Processes often need to work together or at the very least share resources.

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

Inter-Process Communication

Race Conditions:

result depends on exact order of processes running



Process will first inquire about free slot (read), then write to the free slot and finally increment the free slot index.

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

Intra Process Communication

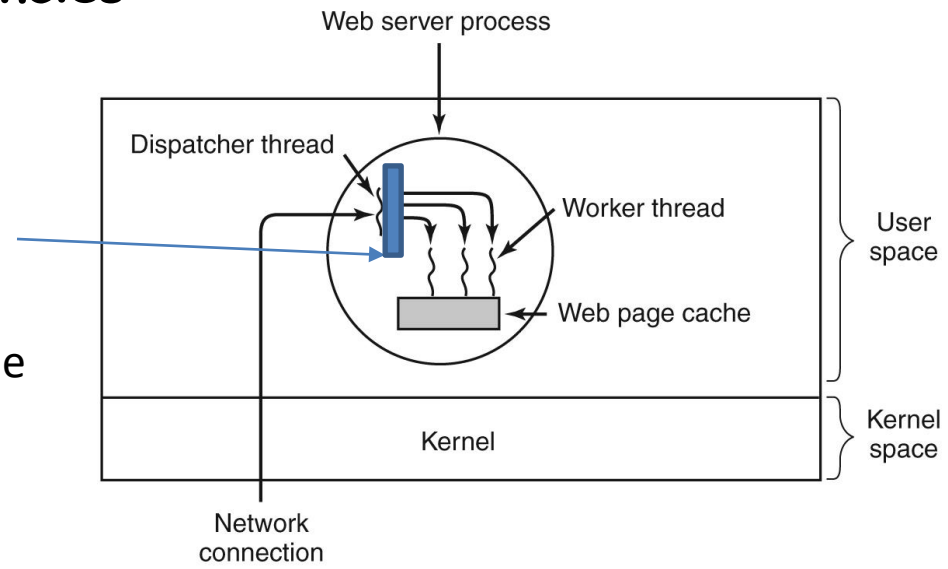
- Threads often need to work together and access resources (e.g. memory) in their common address space.

Same Issues:

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

Example: Multi-threaded Webserver

- Dispatcher thread deposits work into queue of requests to be processed
- Worker threads will pick work from the queue



Read-Modify-Write Cycles

- For instance, you read a variable , make a decision and modify the variable
- Examples:

read

modify-write

race or preemption

```
If ( var == 0 ) {
        /*do something */
        var = 1;
}
```

```
If ( var == 0 ) {
        var = 1;
        /*do something */
}
```

```
    ldw r3, @var
    cmpwi r3, 0
    bne L1
    stwi @var, 1
    /* some code */
L1:
```

- These will have problems if the code is executed by two threads concurrently that accesses the same data

  ➔ Read-Modify-Write cycles are typically an issue

- Consider the race and preemption case with 2 threads

Expectation is that code has "consistent view" at data

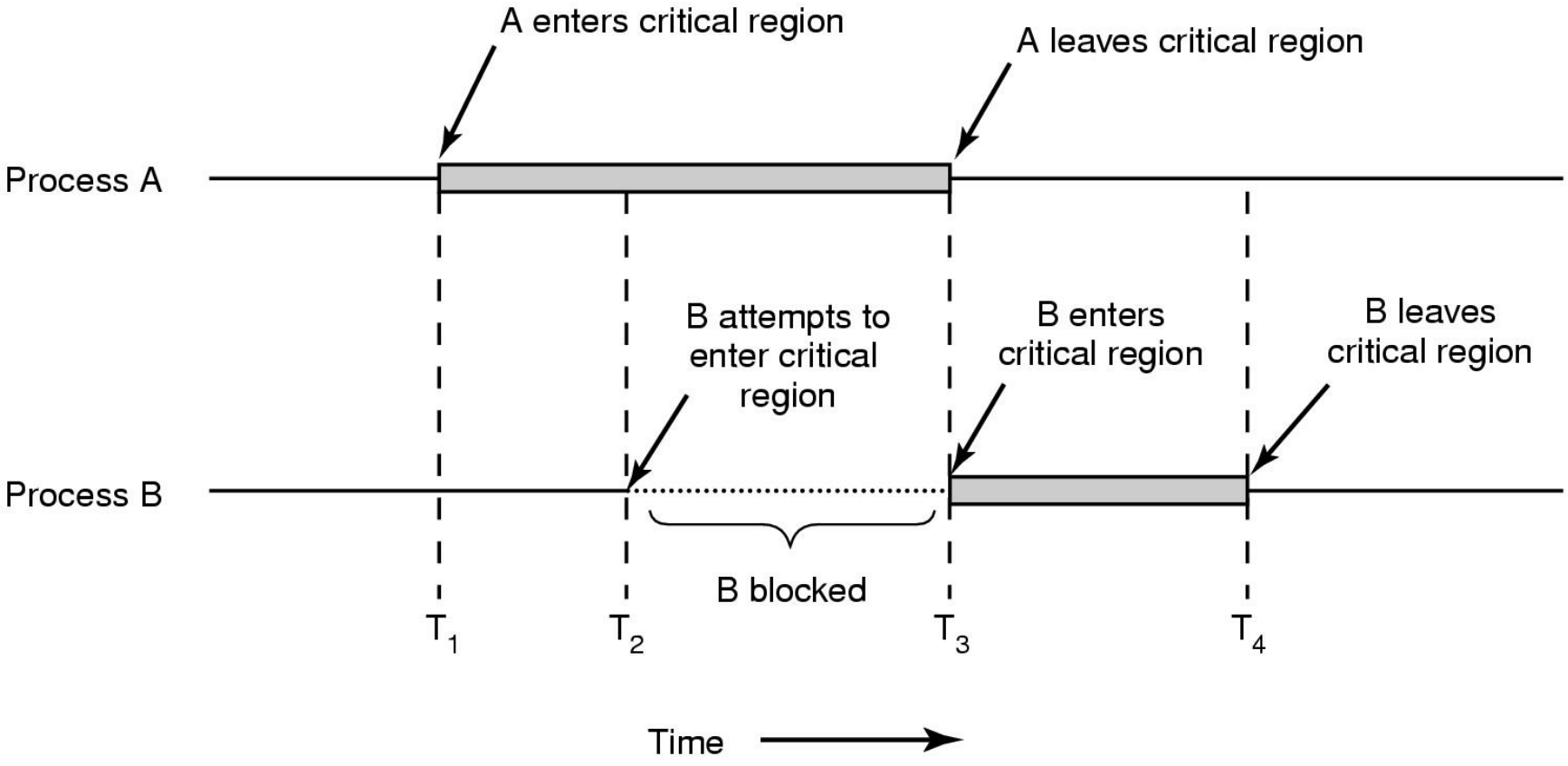
Critical Regions

Mutual Exclusion: Only one process (thread) 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.
  - Won't work in SMP (multi-core systems)

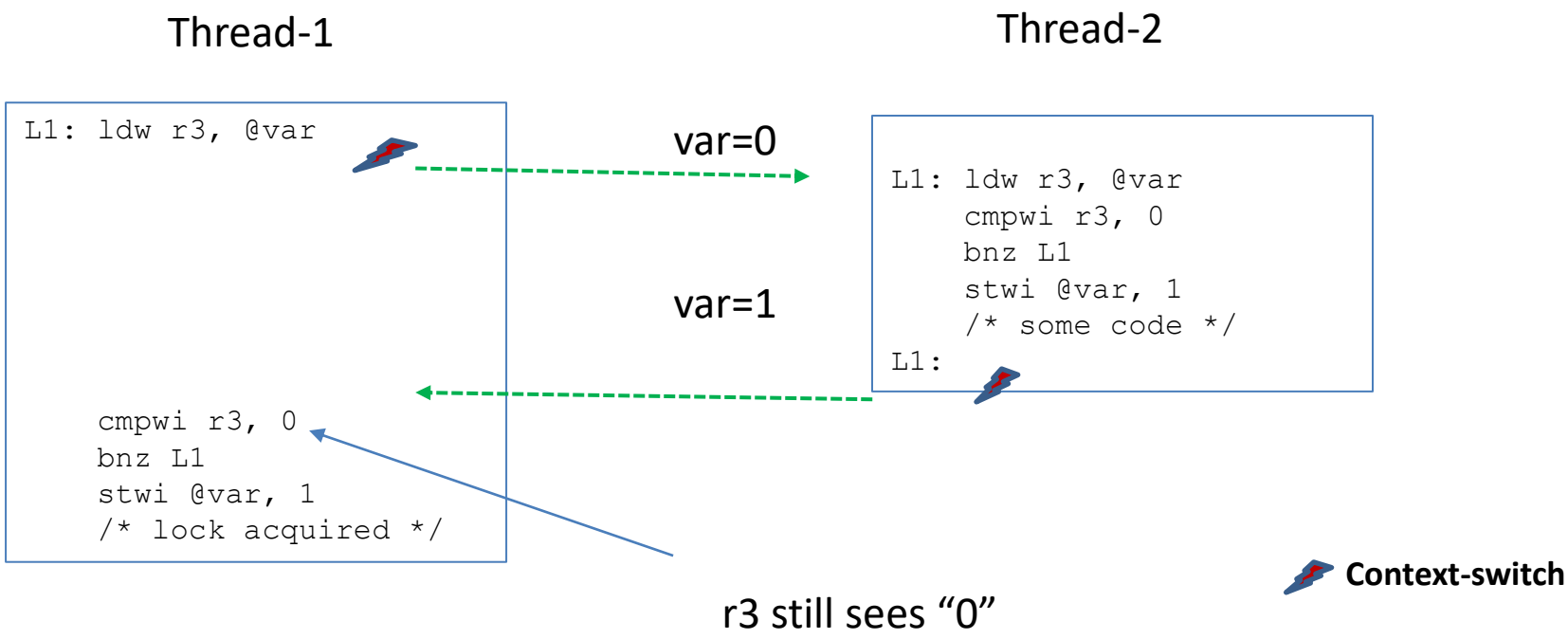
Lock variable?

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

  --> READ-MODIFY-WRITE cycle

Read-Modify-Write Cycles During Locks

- Using a simple Lock Variable <var>



- Both Threads assume now they hold the lock

Mutual Exclusion with Busy Waiting

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

(a)
Process 0

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

(b)
Process 1

Proposed solution to critical region problem

But that's just a toy and in reality not useful

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
```

Entering and leaving a critical region using the TSL instruction.

The XCHG Instruction (other arch have cmp_and_swap)

```
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.

Load/Store Conditional

- Modern processors resolve cmp_and_swap through the cache coherency and instructions
- Reservation (ldwx) remembers ONE address on a CPU and verifies on (stwx) whether it still holds reservation, otherwise store will fail
- Reservation is lost on interrupts or if another CPU steals cacheline holding <lockvar> (see discussion in first lecture on cache coherency) or if another ldwx is issued.
- Can't do complicated things and particular no stores

```
void atomic_inc(unsigned long* var)
{
  L1:  ldwx r3, @var      // load r3 and set reservation register on CPU with &var
       add r3,r3,1        // r3 = r3+1
       stwx r3, @var      // store conditionally r3 back to lock if reservation is still held
       bcond L1           // if store conditionally failed try again
}
```

Solutions so far

- Both TSL (xchg/cmpswp/ldst-cond) and Peterson's solutions are correct, but they
  - rely on busy-waiting during "contention"
  - waste CPU cycles
  - Priority Inversion problem

Priority Inversion Problem

- Higher priority process can be prevented from entering a critical section (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

Lock/Mutex Implementation with semi busy waiting

```
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
```

Thread gives up the CPU and upon reschedule it will attempt again.

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

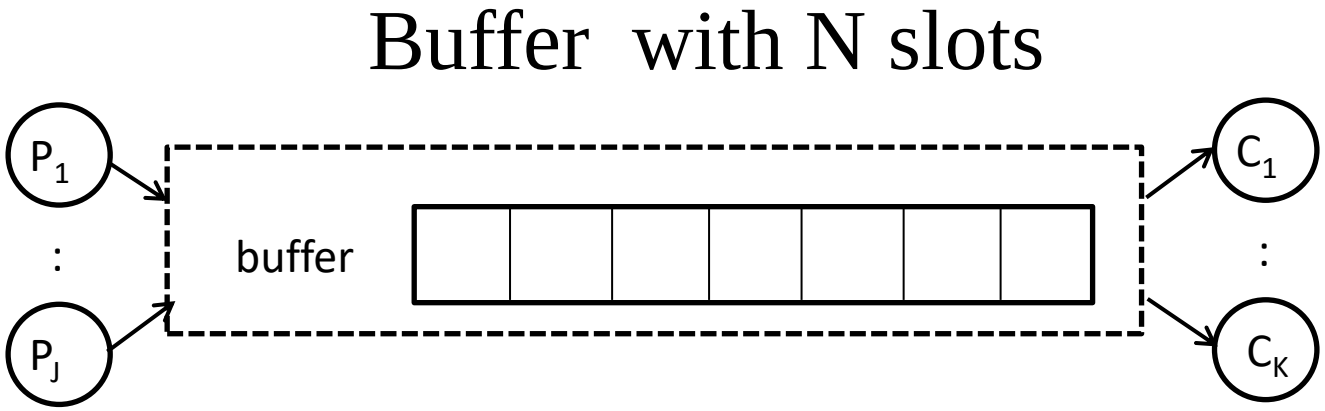
Lock Contention

- Lock Contention arises when a process/thread attempts to acquire a lock and the lock is not available.

- This is a function of
  - frequency of attempts to acquire the lock
  - Lock hold time (time between acquisition and release)
  - Number of threads/processes acquiring a lock

- Lock contention is a function of lock hold time and lock acquisition frequency.
- If lock contention is low, TSL is an OK solution.
- The linux kernel uses it extensively for many locking scenarios.
- Alternative approaches will be discussed next.

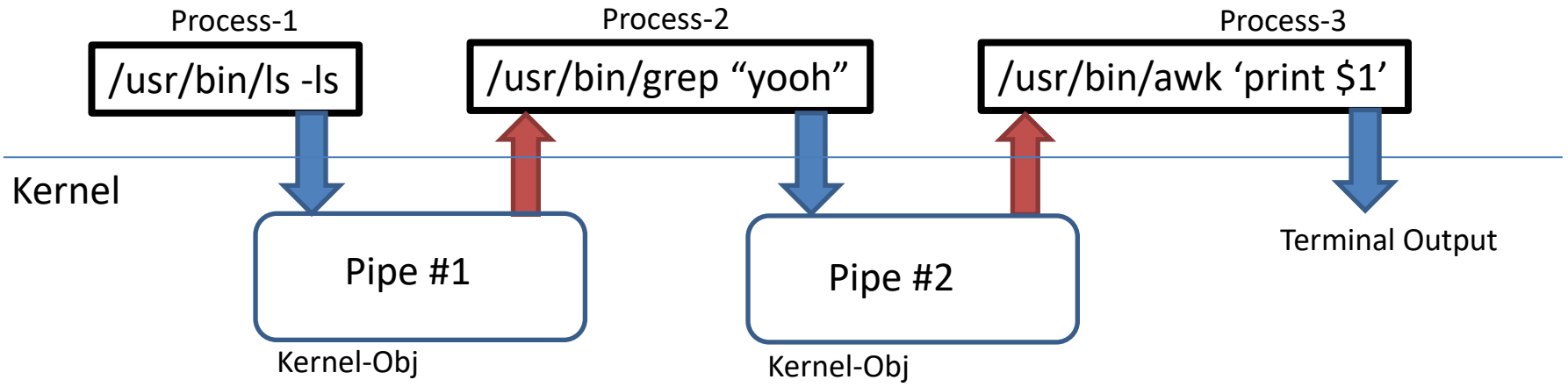
Producer-Consumer problem



- Producer: (1 .. j)
  - How do I know a slot is free ?
  - How do I know a slot became free ?
- Consumer: (1 .. k)
  - 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 */
     }
}
```

Not sure why the book put a while loop here, we just assume that these function produce/consume one elem

```
#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)

Requirements

- Need a mechanism that allows synchronization between processes/threads on the base of shared resource.
- Synchronization implies interaction with scheduling sub-system.
- Led to the innovation of semaphores.

Semaphore Data Structure

Dijkstra 1965

```
class Semaphore
{
     int  value;                 // counter
     Queue<Thread*> waitQ  ; // queue of threads blocked
                                 // this sema
     void Init(int v);          // initialization
     void P();                  // down(), wait()
     void V();                  // up(), signal ()
}
```

Initially developed as a "construct" to ease programming.

This version is based on multi-threaded capable OS or thread library
Older version used Process as the object , same principle

Semaphore implementations

```
void Semaphore::Init(int v)
{
     value = v;
     waitQ.init(); // empty queue
}
```

Semaphore implementations

```
void Semaphore::P() //  or   wait()  or down()
{
     value = value - 1;
     if (value < 0)
     {
          waitQ.add(current_thread);
          current_thread->status = blocked;
          schedule();  // forces wait, thread blocked
     }
}
```

AKA "acquiring or grabbing the semaphore"

think of it as obtaining access rights

Semaphore implementations

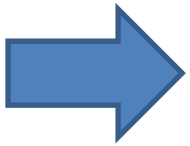
```
void Semaphore::V() //   or   signal()   or up()
{
    value = value + 1;
    if (value <= 0)
    {
       Thread *thd = waitQ.getNextThread();

       scheduler->add(thd); // make it scheduable
    }
}
```

AKA "releasing the semaphore"

Semaphore Solution

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



P() and V() must be atomic.

e.g.

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

However: disabling interrupts only works on single CPU systems
Atomic lock variable an option on entry and exit,
but must release the lock as part of call to schedule() in P()
or must reacquire the lock as part of call to schedule() in V()

- Use lock variable with atomic op

Semaphore Data Structure (with atomicity)

```
class Semaphore
{
     int  lockvar;              // to guarantee atomicity
     int  value;                // counter
     Queue<Thread*> waitQ;      // queue of threads
                                //  waiting on this sema
     void Init(int v);  // initialization
     void P();          // down(), wait()
     void V();          // up(), signal ()
}
```

Semaphore implementations

```
void Semaphore::P() //  or   wait()  or down()
{
     lock(&lockvar);
     value = value - 1;
     if (value < 0)
     {
          waitQ.add(current_thread);
          current_thread->status = blocked;
          unlock(&lockvar);
          schedule();  // forces wait, thread block
     } else {
          unlock(&lockvar);
     }
}
```

Semaphore implementations

```
void Semaphore::V() //   or   signal()   or up()
{
    lock(&lockvar);
    value = value + 1;
    if (value <= 0)
    {
       Thread *thd = waitQ.getNextThread();

       scheduler->add(thd); // make it scheduable
    }
    unlock(&lockvar);
}
```

Two kinds of semaphores

- Mutex semaphores
  (or binary semaphores or LOCK):
  for mutual exclusion problems:
  value initialized to 1

- Counting semaphores:
  for synchronization problems.
  Value initialized to any value 0..N
  Value shows available tokens to enter or number or processes waiting when negative.

Semaphore Solution To the Producer Consumer Problem

```
#define N <somenumber>
Semaphore empty = N;
Semaphore full  = 0;
Semaphore mutex = 1;
T buffer[N];
int widx = 0, ridx = 0;
```

3 Semaphores required

LOCK

signaling →

```
Producer(T item)
{
        P(&empty);

        P(&mutex);      // Lock

        buffer[widx] = item;
        widx = (widx + 1 ) % N;

        V(&mutex);      // Unlock

        V(&full);
}
```

```
Consumer(T &item)
{
        P(&full);

        P(&mutex);      // Lock

        item = buffer[ridx];
        ridx = (ridx + 1 ) % N;

        V(&mutex);      // Unlock

        V(&empty);
}
```

Semaphore Solution To the Producer Consumer Problem

```
#define N <somenumber>
Semaphore empty   = N;
Semaphore full    = 0;
Semaphore mutex_w = 1;
Semaphore mutex_r = 1;
T buffer[N];
int widx = 0, ridx = 0;
```

4 Semaphores for lower lock contention

LOCK

signaling

```
Producer(T item)
{
        P(&empty);

        P(&mutex_w);     // Lock

        buffer[widx] = item;
        widx = (widx + 1 ) % N;

        V(&mutex_w);     // Unlock

        V(&full);
}
```

```
Consumer(T &item)
{
        P(&full);

        P(&mutex_r);     // Lock

        item = buffer[ridx];
        ridx = (ridx + 1 ) % N;

        V(&mutex_r);     // Unlock

        V(&empty);
}
```

Semaphore Solution To the Producer Consumer Problem

```
#define N <somenumber>
Semaphore empty   = N;
Semaphore full    = 0;
Semaphore mutex_w = 1;
Semaphore mutex_r = 1;
T buffer[N];
int widx = 0, ridx = 0;
```

4 Semaphores for lower lock contention

LOCK

signaling →

This example doesn't work; too aggressive !!!!!

```
Producer(T item)
{
        P(&empty);

        P(&mutex_w);      // Lock

        int wi = wdix;
        widx = (widx + 1 ) % N;

        V(&mutex_w);      // Unlock

        buffer[wi] = item;

        V(&full);
}
```

Nasty race condition on wi

```
Consumer(T &item)
{
        P(&full);

        P(&mutex_r);      // Lock

        int ri = rdix;
        ridx = (ridx + 1 ) % N;

        V(&mutex_r);      // Unlock

        item = buffer[ri];

        V(&empty);
}
```

Shorter lock hold times by only protecting the indices not the buffer themselves

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);
}
```

Using threads to solve the producer-consumer problem.

Problems with semaphore?

- It can be difficult to write semaphore code (arguably)
- One has to be careful in code construction
- If a thread dies and it holds a 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 acquire multiple locks in the same order

- *Preferably* release in reverse order as acquired: not required but good hygiene

  lots of discussion on this, there are scenarios where that is not desired but highly optimized implementation

- Example: SMP CPU scheduler where load balancing is required.

Example

```
doit()
{
   s1.P();
   s2.P();
   "do something awesome"
   s1.V();
   s2.V();
}
```

OK

```
doit()
{
   s1.P();
   s2.P();
   "do something awesome"
   s2.V();
   s1.V();
}
```

Better

```
doit()
{
   s1.P();
   {
       s2.P();
       "do something awesome"
       s2.V();
   }
   s1.V();
}
```

Making nesting clearer

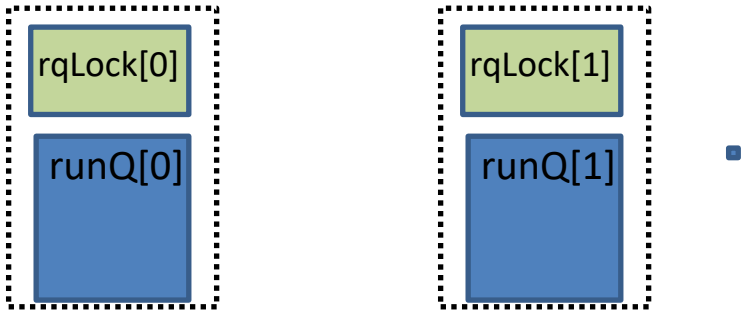
Example

```
doit()
{
  s1.P();
  do {
      s2.P();
      do {
          "do something awesome"
          break;
          "do something else awesome"
      } while(0);
      s2.V();
  } while (0);
  s1.V();
}
```

Making nesting clearer

Example: SMP Scheduler

- Each cpu_i has (runQ[i] and rqLock[i])




```
Schedule(int i)
{
  lock(rqlock[i]);
  :
  { // load balance with j
      lock(rqlock[j]);
      ; // pull some threads
      unlock(rqlock[j]);
  }
  :
  unlock(rqlock[i]);
}
```

- cpu0 (i=0, j=1) ; cpu1 (i=1, j=0)

- Can lead to deadlocks → must use same order

Example SMP Scheduler

- So how do we get correct order?
- We force it !!
  if necessary release owned lock first and reaquire.

```
Schedule(int i)
{
  lock(rqlock[i]);
  :
  { // load balance with j
      add_lock(i,j);
      ; // pull some threads
      unlock(rqlock[j]);
  }
  :
  unlock(rqlock[i]);
}
```

```
add_lock(int hlv, int alv)
{
  if ( hlv > alv ) {
     // first unlock hlv
     unlock(rqlock[hlv]);
     lock(rqlock[alv];
     lock(rqlock[hlv];
  } else {
     lock(rqlock[alv]);
  }
}
```

- We just need some order (could be "<" or based on addresses, it doesn't matter)

So which lock should I use

- Busy Lock vs. Semaphores ?
- If lock hold time is short and/or code is uninterruptible, then lock variables and busy waiting are OK (linux kernel uses it all the time)
- Otherwise use semaphores

Other Unix/Linux Mechanisms

- File based:

  `flock()`

- System V semaphores
  heavy weight as each call is a system call going into the kernel

  `semget(), semop()` [P and V]

- Futexes

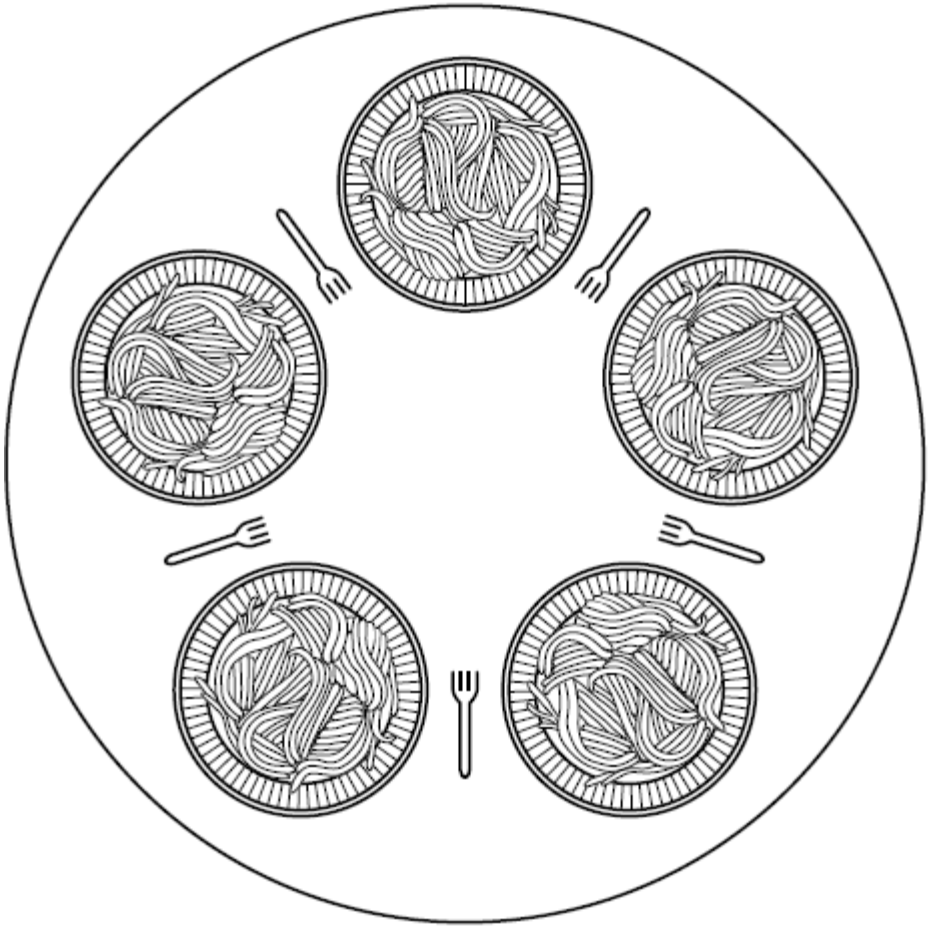
  Lighter weight as uncontested cases are resolved done using cmpxchg in userspace and if race condition is recognized it goes to kernel

  `futex()`

- Message queues

  `mq_open(),mq_close(),mq_send(), mq_receive()`

The Dining Philosophers Problem (1)



Lunch time in the Philosophy Department.

The Dining Philosophers Problem (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                       /* philosopher is thinking */
                take_fork(i);                   /* take left fork */
                take_fork((i+1) % N);           /* take right fork; % is modulo operator */
                eat( );                         /* yum-yum, spaghetti */
                put_fork(i);                    /* put left fork back on the table */
                put_fork((i+1) % N);            /* put right fork back on the table */
        }
}
```

A nonsolution to the dining philosophers problem.

The Dining Philosophers Problem (3)

```
#define N           5                       /* number of philosophers */
#define LEFT        (i+N–1)%N               /* number of i's left neighbor */
#define RIGHT       (i+1)%N                 /* number of i's right neighbor */
#define THINKING    0                       /* philosopher is thinking */
#define HUNGRY      1                       /* philosopher is trying to get forks */
#define EATING      2                       /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                          /* repeat forever */
        think( );                           /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat( );                             /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}
```

A solution to the dining philosophers problem.

The Dining Philosophers Problem (4)

```
            put_forks(i);                    /* put both forks back on table */
        }
    }

    void take_forks(int i)                   /* i: philosopher number, from 0 to N–1 */
    {
        down(&mutex);                        /* enter critical region */
        state[i] = HUNGRY;                   /* record fact that philosopher i is hungry */
        test(i);                             /* try to acquire 2 forks */
        up(&mutex);                          /* exit critical region */
        down(&s[i]);                         /* block if forks were not acquired */
    }

    void put_forks(i)                        /* i: philosopher number, from 0 to N–1 */
```

A solution to the dining philosophers problem.

The Dining Philosophers Problem (5)

```
}

void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                      /* enter critical region */
        state[i] = THINKING;               /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

A solution to the dining philosophers problem.

The Multiple Readers and Writer Problem (1)

It should be allowed for multiple readers to be active on a data as long as the data is not modified

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read( );               /* noncritical region */
    }
}
```

void writer(void)

A solution to the multiple readers and writer problem.

The Multiple Readers and Writer Problem (1)

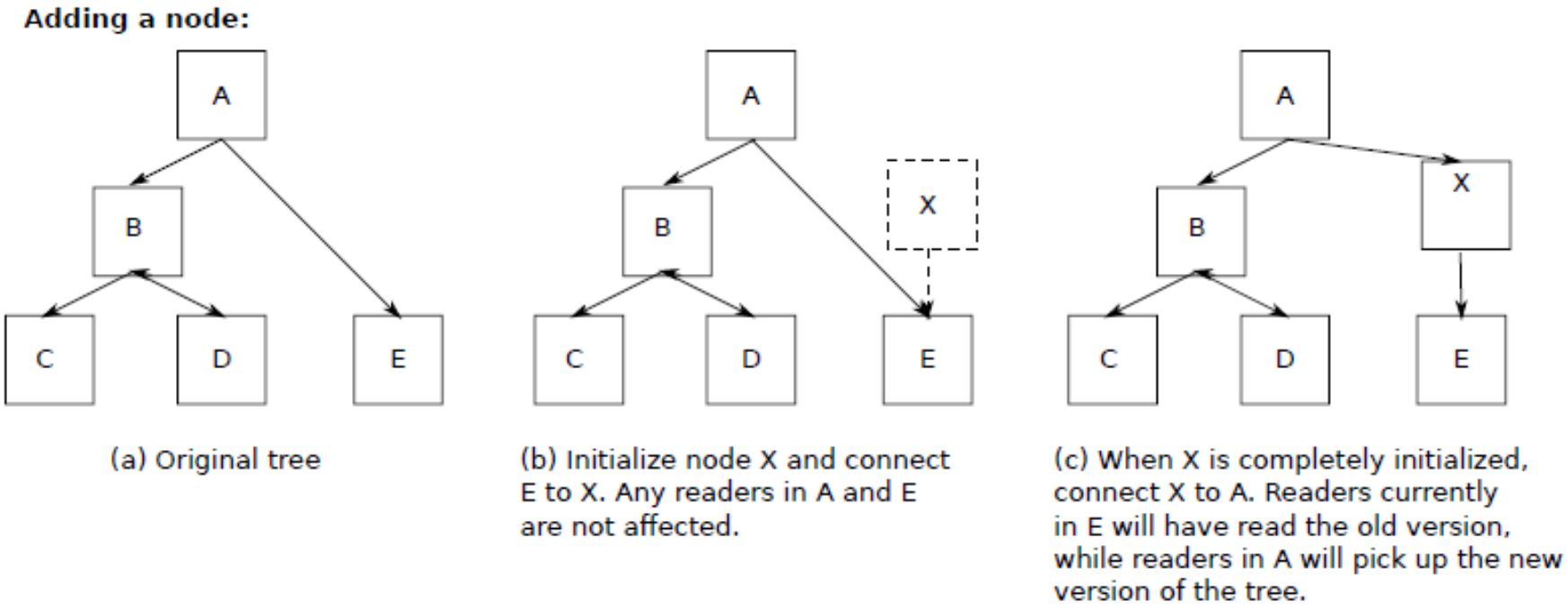
Only one writer can be active but then no reader can be active
Writer starvation still possible

```
            use_data_read( );           /* noncritical region */
        }
}


void writer(void)
{
        while (TRUE) {                  /* repeat forever */
            think_up_data( );           /* noncritical region */
            down(&db);                  /* get exclusive access */
            write_data_base( );         /* update the data */
            up(&db);                    /* release exclusive access */
        }
}
```

A solution to the multiple readers and writer problem.

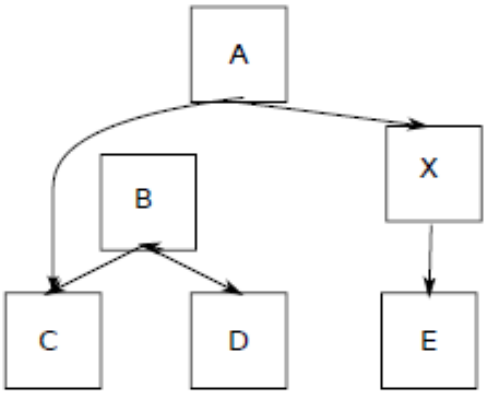
Avoiding Locks: Read-Copy-Update (1)



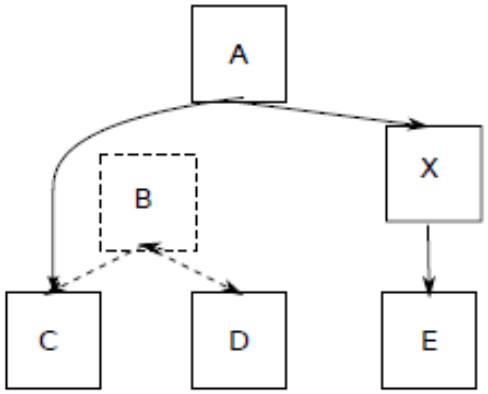
Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)

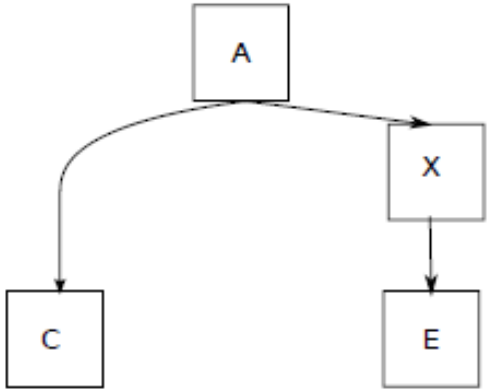
Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.



(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.



(f) Now we can safely remove B and D

Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

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.

Monitor Example

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

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

Figure 2-33. A monitor.

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**?

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

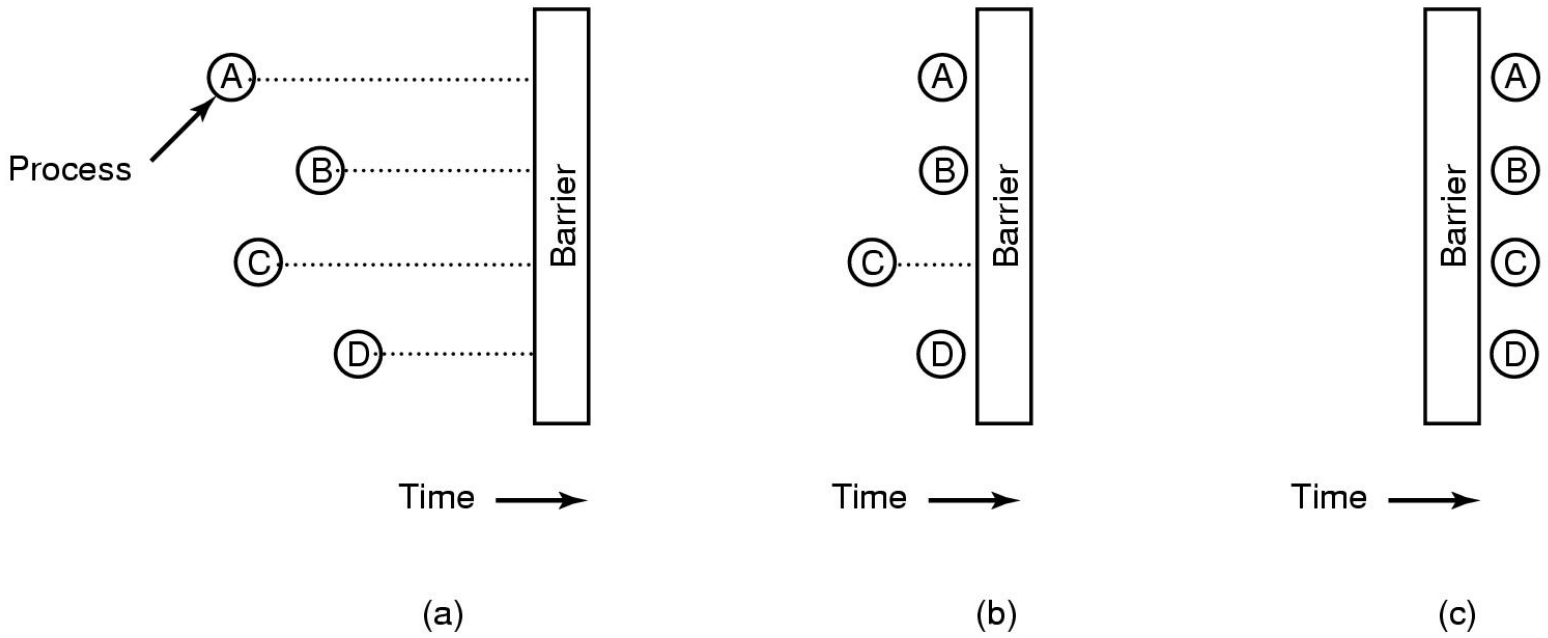


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Deadlock vs Starvation

- Deadlock – Process(es) waiting on events (resources) that will never happen:
  - Can be system wide or just one process
    [will define this more precisely later]

- 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 Nonpreemptable 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-threaded 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.

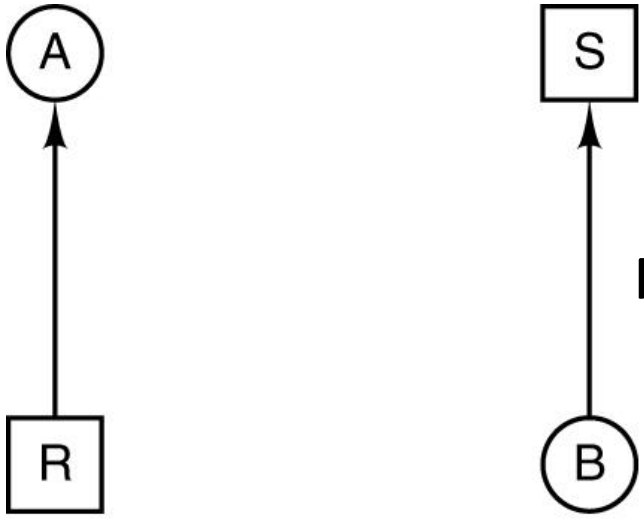
Resource Allocation Graph

Process

Process A is holding resource R

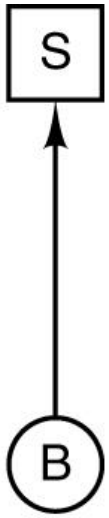
Resource

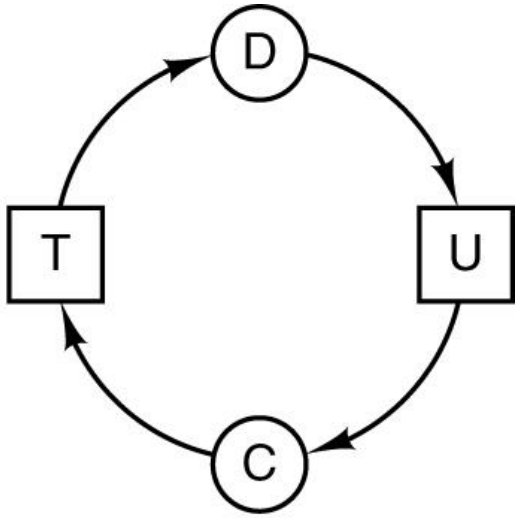
Resource Allocation Graph



Process B is requesting resource S

Resource Allocation Graph





Deadlock!

A

Request R
Request S
Release R
Release S

(a)

B

Request S
Request T
Release S
Release T

(b)

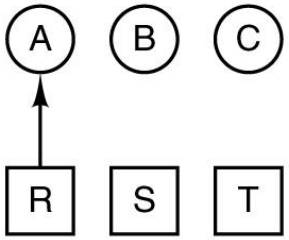
C

Request T
Request R
Release T
Release R

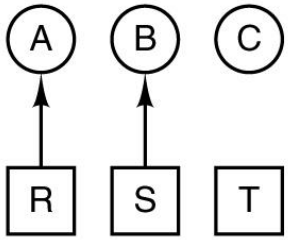
(c)

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

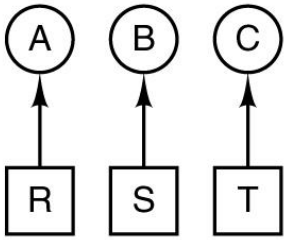
(d)



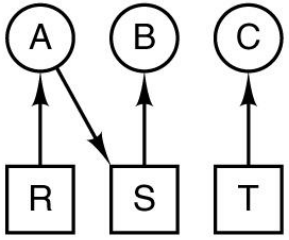
(e)



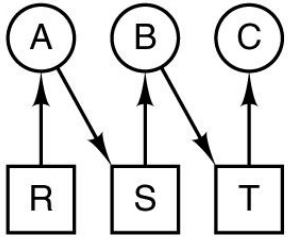
(f)



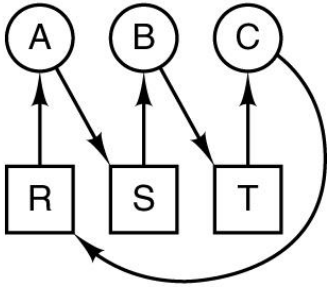
(g)



(h)



(i)



(j)

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

However: remember Murphy's law !!!

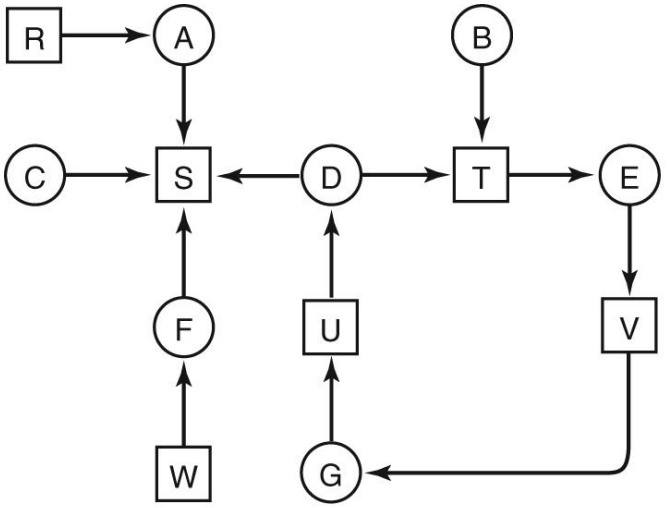
What can go wrong, will ……

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

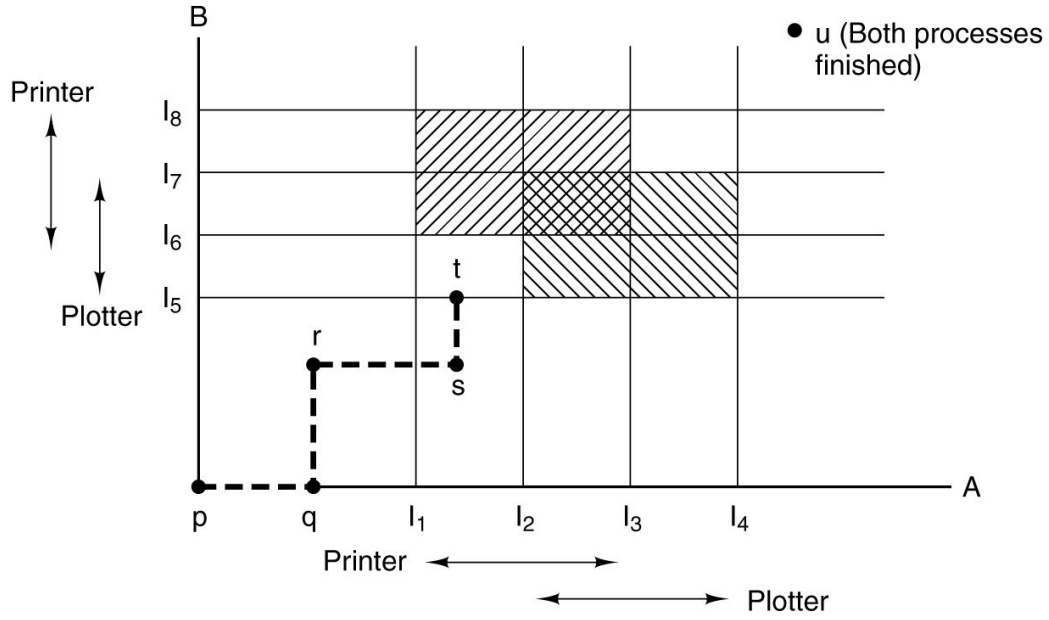
- Have processes checkpointed periodically
- Checkpoint of a process: its state is written to a file so that it can be restarted later
- In case of deadlock, a process that owns a needed resource is rolled back to the point before it acquired that resource

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



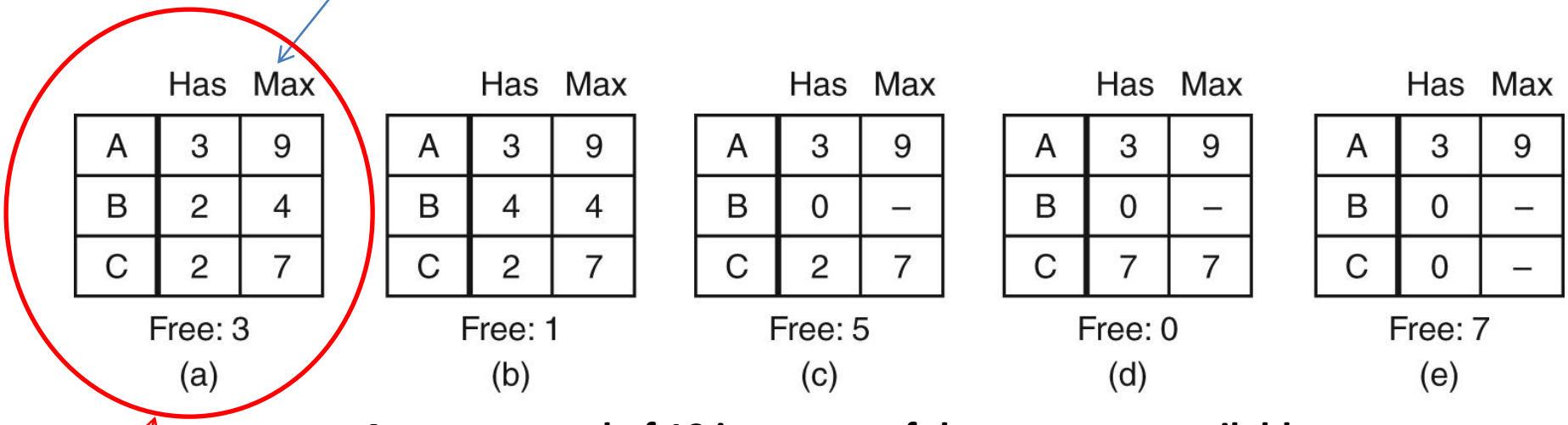
Safe and Unsafe States

- 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

Conclusions

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