Interprocess Communication
and
Synchronization

Chester Rebeiro
IIT Madras

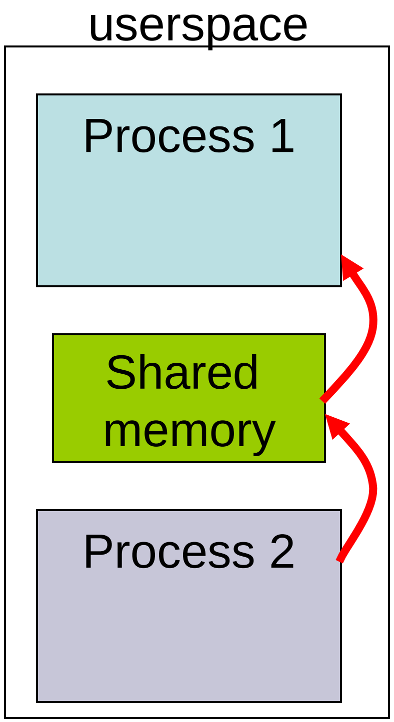
Inter Process Communication

- Advantages of Inter Process Communication (IPC)
  - Information sharing
  - Modularity/Convenience
- 3 ways
  - Shared memory
  - Message Passing

  - Signals

Shared Memory

- One process will create an area in RAM which the other process can access
- Both processes can access shared memory like a regular working memory
  - Reading/writing is like regular reading/writing
  - Fast
- Limitation : Error prone. Needs synchronization between processes

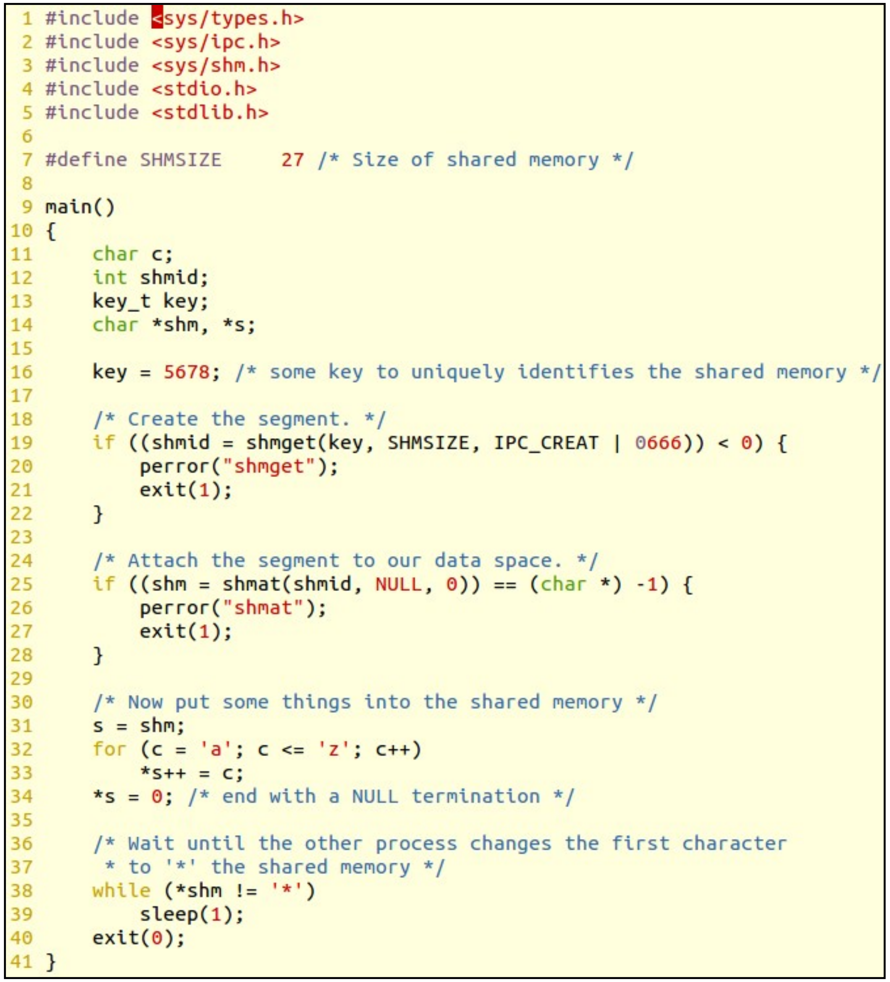


Shared Memory in Linux

- **int shmget (key, size, flags)**
  - Create a shared memory segment;
  - Returns ID of segment : shmid
  - key : unique identifier of the shared memory segment
  - size : size of the shared memory (rounded up to the PAGE_SIZE)
- **int shmat(shmid, addr, flags)**
  - **At**tach shmid shared memory to address space of the calling process
  - addr : pointer to the shared memory address space
- **int shmdt(shmid)**
  - **D**e**t**ach shared memory

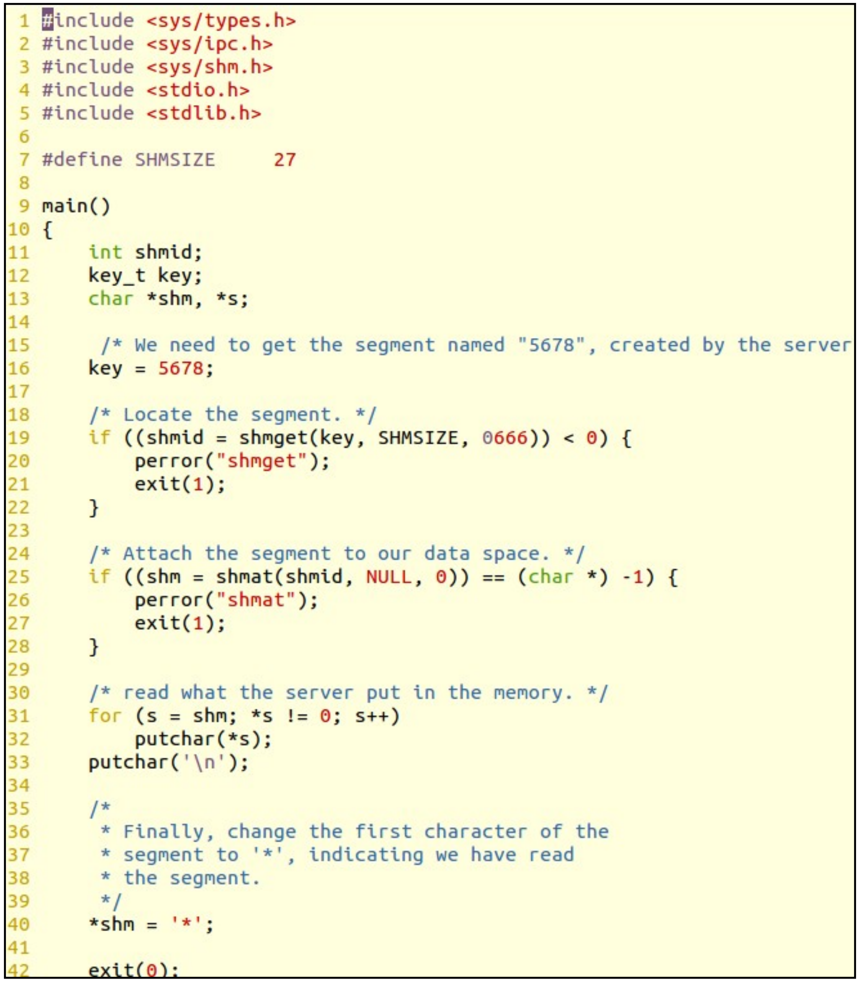
Example

server.c



```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27 /* Size of shared memory */

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* some key to uniquely identifies the shared memory */

    /* Create the segment. */
    if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now put some things into the shared memory */
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = 0; /* end with a NULL termination */

    /* Wait until the other process changes the first character
     * to '*' the shared memory */
    while (*shm != '*')
        sleep(1);
    exit(0);
}
```

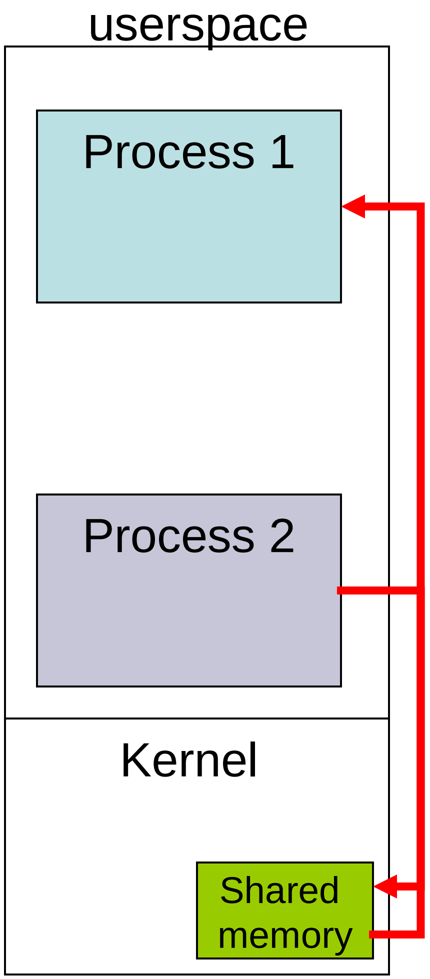
client.c



```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define SHMSIZE     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

     /* We need to get the segment named "5678", created by the server
    key = 5678;

    /* Locate the segment. */
    if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* read what the server put in the memory. */
    for (s = shm; *s != 0; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
```

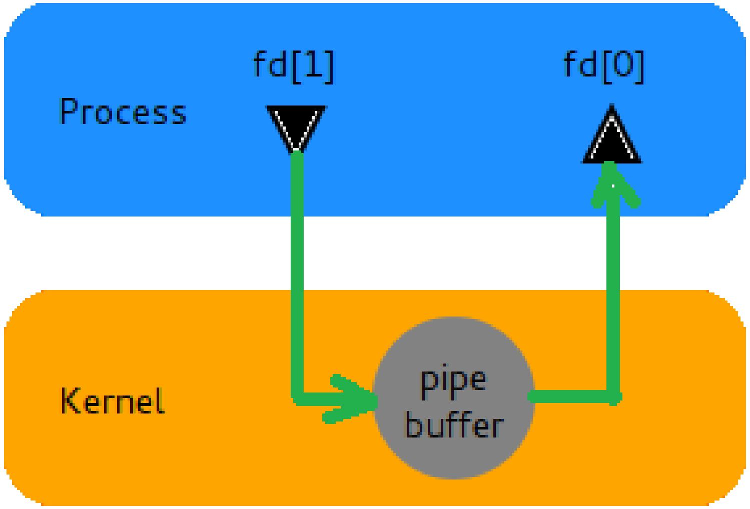
Message Passing

- Shared memory created in the kernel
- System calls such as send and receive used for communication
  - Cooperating : each send must have a receive
- Advantage : Explicit sharing, less error prone
- Limitation : Slow. Each call involves marshalling / demarshalling of information

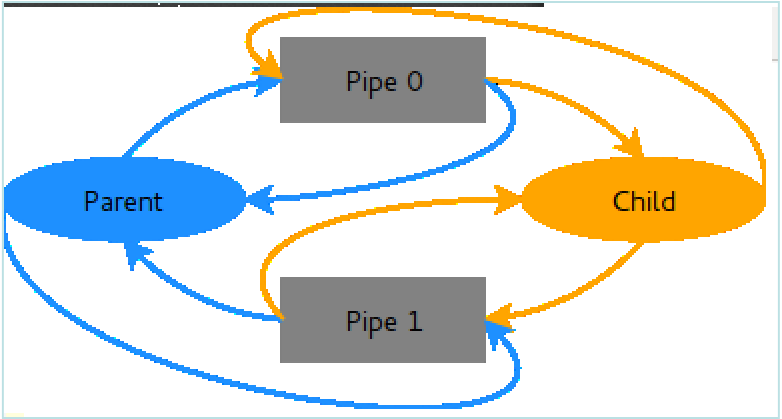


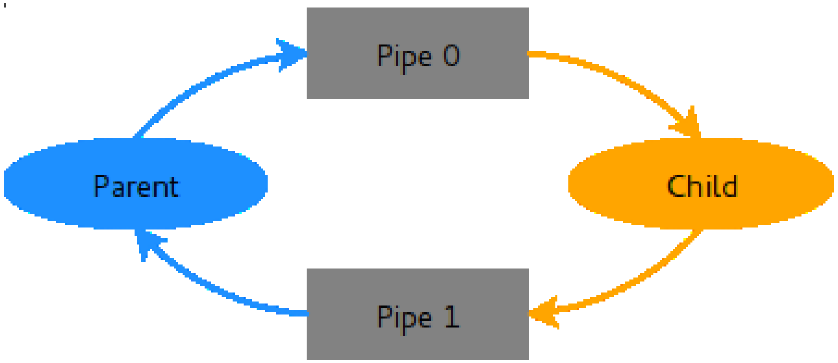
Pipes

- Always between parent and child
- Always unidirectional
- Accessed by two associated file descriptors:
  - fd[0] for reading from pipe
  - fd[1] for writing to the pipe



Pipes for two way communication





- Two pipes opened

  pipe0 and pipe1
- Note the unnecessary pipes

- Close the unnecessary pipes

Example

(child process sending a string to parent)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
  int pipefd[2];
  int pid;
  char recv[32];

  pipe(pipefd);

  switch(pid=fork()) {
  case -1: perror("fork");
           exit(1);
  case 0:                                    /* in child process */
        close(pipefd[0]);                    /* close unnecessary pipefd */
        FILE *out = fdopen(pipefd[1], "w"); /* open pipe descriptor as stream */
        fprintf(out, "Hello World\n");       /* write to out stream */
        break;
  default:                                   /* in parent process */
        close(pipefd[1]);                    /* close unnecessary pipefd */
        FILE *in = fdopen(pipefd[0], "r");  /* open descriptor as stream */
        fscanf(in, "%s", recv);              /* read from in stream */
        printf("%s", recv);
        break;
  }
}
```

Signals

- Asynchronous unidirectional communication between processes
- Signals are a small integer
  - eg. 9: kill, 11: segmentation fault
- Send a signal to a process
  - `kill(pid, signum)`
- Process handler for a signal
  - `sighandler_t signal(signum, handler);`
  - Default if no handler defined

ref : http://www.comptechdoc.org/os/linux/programming/linux_pgsignals.html

Synchronization

Chester Rebeiro
IIT Madras

Motivating Scenario

shared variable

```
int counter=5;
```

program 0

```
{
   *
   *
counter++
   *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Single core
  - Program 1 and program 2 are executing at the same time but sharing a single core

Motivating Scenario

program 0

```
{
   *
   *
counter++
   *
}
```

Shared variable

```
int counter=5;
```

program 1

```
{
   *
   *
counter--
   *
}
```

- What is the value of counter?
  - – expected to be 5
  - – but could also be 4 and 6

Motivating Scenario

Shared variable

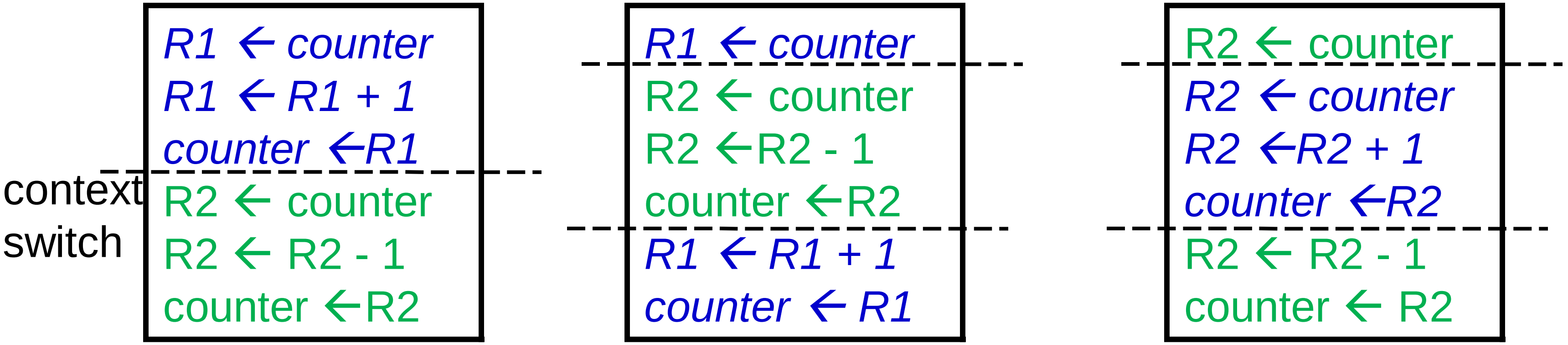
```
int counter=5;
```

program 0

```
{
   *
   *
counter++
   *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```



context switch

R1 ← counter
R1 ← R1 + 1
counter ←R1
R2 ← counter
R2 ← R2 - 1
counter ←R2

counter = 5

R1 ← counter
R2 ← counter
R2 ←R2 - 1
counter ←R2
R1 ← R1 + 1
counter ← R1

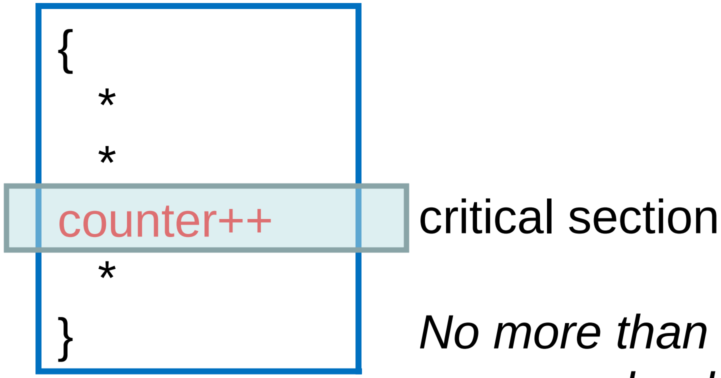
counter = 6

R2 ← counter
R2 ← counter
R2 ←R2 + 1
counter ←R2
R2 ← R2 - 1
counter ← R2

counter = 4

Race Conditions

- Race conditions
    - A situation where several processes access and manipulate the same data (*critical section*)
    - The outcome depends on the order in which the access take place
    - Prevent race conditions by synchronization
        - Ensure only one process at a time manipulates the critical data



```
{
   *
   *
 counter++
   *
}
```

critical section

No more than one process should execute in critical section at a time

Race Conditions in Multicore

shared variable

```
int counter=5;
```

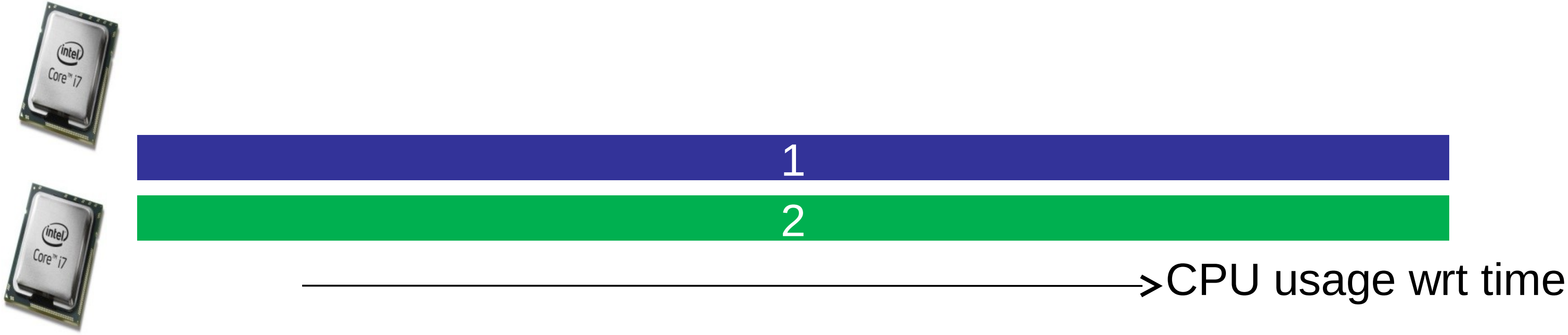
program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Multi core
  - Program 1 and program 2 are executing at the same time on different cores



Critical Section

- Requirements
  - **Mutual Exclusion :** No more than one process in critical section at a given time
  - **Progress :** When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay
  - **No starvation (bounded wait):** There is an upper bound on the number of times a process enters the critical section, while another is waiting.

Locks and Unlocks

program 0

```
{
  *
  *
lock(L)
counter++
unlock(L)
  *
}
```

shared variable

```
int counter=5;
lock_t L;
```

program 1

```
{
  *
  *
lock(L)
counter--
unlock(L)
  *
}
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

When to have Locking?

- Single instructions by themselves are atomic

  eg. add %eax, %ebx

- Multiple instructions need to be explicitly made atomic
  - Each piece of code in the OS must be checked if they need to be atomic

How to Implement Locking

Using Interrupts

Process 1

```
      while(1){
lock--->  disable interrupts ()
          critical section
unlock--->enable interrupts ()
           other code
      }
```

Process 2

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
     other code
}
```

- Simple
  - When interrupts are disabled, context switches won’t happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Software Solution (Attempt 1)

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2;  // unlock
     other code
}
```

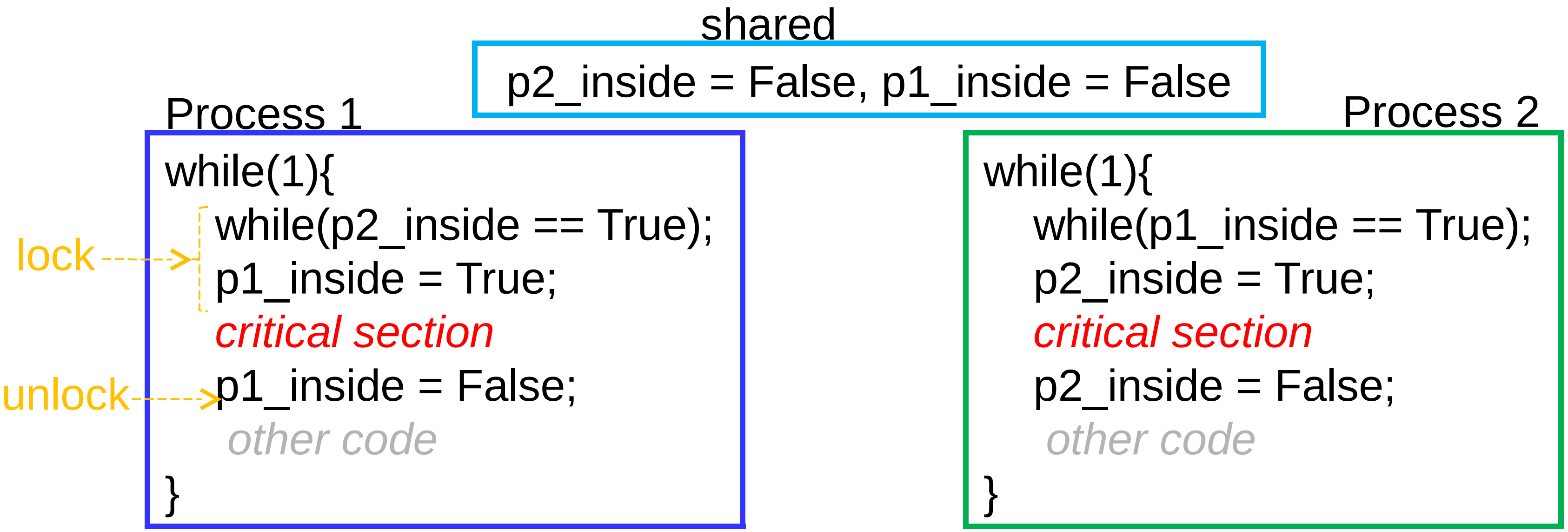
Process 2

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

- Achieves mutual exclusion
- Busy waiting – waste of power and time
- Needs to alternate execution in critical section

  process1 →*process2* →*process1* →*process2*

Software Solution (Attempt 2)



shared

```
p2_inside = False, p1_inside = False
```

Process 1

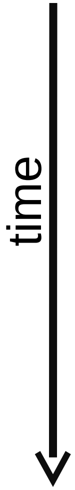
```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
     other code
}
```

Process 2

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
     other code
}
```

- Need not alternate execution in critical section
- Does not guarantee mutual exclusion

Attempt 2: No mutual exclusion



CPU	**p1_inside**	**p2_inside**
while(p2_inside == True);	False	False
context switch		
while(p1_inside == True);	False	False
p2_inside = True;	False	True
context switch		
p1_inside = True;	True	True

Both p1 and p2 can enter into the critical section at the same time

Software Solution (Attempt 3)

globally defined

```
p2_wants_to_enter, p1_wants_to_enter
```

Process 1

```
      while(1){
           p1_wants_to_enter = True
lock ----> while(p2_wants_to_enter = True);
           critical section
unlock---> p1_wants_to_enter = False
           other code
      }
```

Process 2

```
while(1){
     p2_wants_to_enter = True
     while(p1_wants_to_enter = True);
     critical section
     p2_wants_to_enter = False
     other code
}
```

- Achieves mutual exclusion
- Does not achieve progress (could deadlock)

Attempt 3: No Progress

time

CPU	p1_inside	p2_inside
p1_wants_to_enter = True	False	False
context switch		
p2_wants_to_enter = True	False	False

There is a tie!!!

Both p1 and p2 will loop infinitely

Peterson’s Solution

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
              favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

lock

unlock

If the second process wants to enter. favor it. (be nice !!!)

favored is used to break the tie when both p1 and p2 want to enter the critical section.

(* the process which sets favored last looses the tie *)

Break the tie with a ‘favored’ process

Peterson’s Solution

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
              favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

Process 2

```
while(1){
    p2_wants_to_enter = True
    favored = 1

    while (p1_wants_to_enter AND
              favored = 1);
    critical section
    p2_wants_to_enter = False
    other code
}
```

Bakery Algorithm

- Synchronization between N > 2 processes
- By Leslie Lamport

Eat
when 196 displayed

ZAITOON
IIT CAMPUS
CHENNAI
* COUNTER SALE *
Token No : 196
D: ZA186
NO: CS1
5/09/2015 TIME:6:08:07 PM
RATE QTY AMOUNT

wait your turn!!

http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

Simplified Bakery Algorithm

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

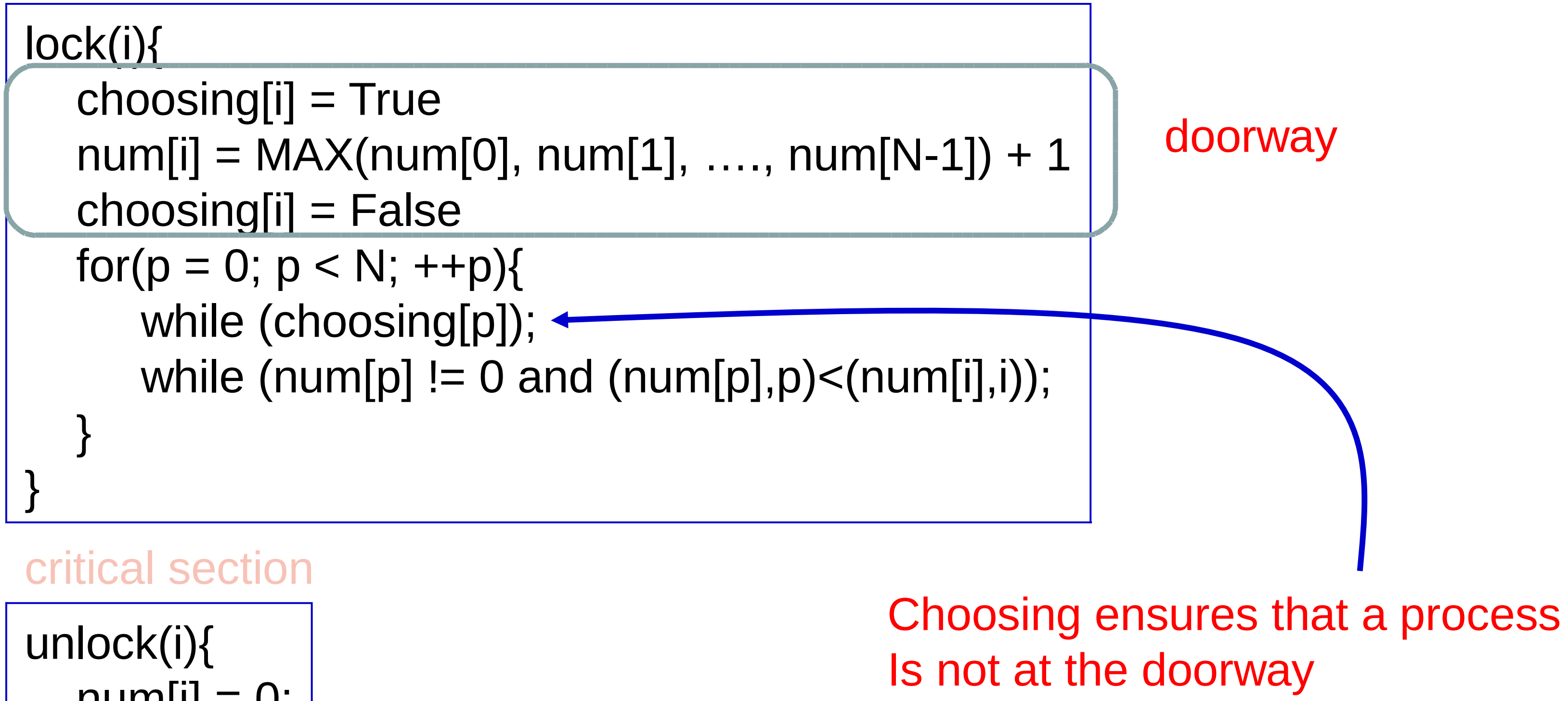
This is at the doorway!!!
It has to be atomic
to ensure two processes
do not get the same token

critical section

```
unlock(i){
    num[i] = 0;
}
```

Original Bakery Algorithm

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.



```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

Choosing ensures that a process
Is not at the doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

Analyze this

- Does this scheme provide mutual exclusion?

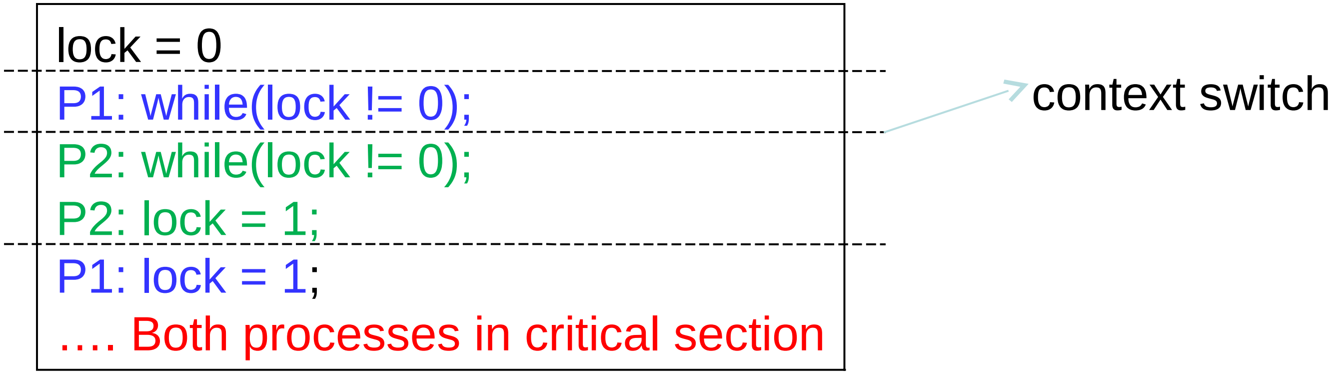
Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Process 2

```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

No



lock = 0

P1: while(lock != 0);

P2: while(lock != 0);

P2: lock = 1;

P1: lock = 1;

…. Both processes in critical section

If only…

- We could make this operation atomic

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

Make atomic

Hardware to the rescue….

Hardware Support
(Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

```
while(1){
    while(test_and_set(&lock) == 1);
    critical section
    lock = 0;  // unlock
    other code
}
```

Usage for locking

Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
So the first invocation of test_and_set will read a 0 and set lock to 1 and return. The second test_and_set invocation will then see lock as 1, and will loop continuously until lock becomes 0

Intel Hardware Software (xchg instruction)

- **xchg** : Intel instruction. exchange.

typical usage :

xchg reg, mem

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

Note. %eax is returned

High Level Constructs

- Spinlock
- Mutex
- Semaphore

Spinlocks Usage

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

- One process will acquire the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

See spinlock.c and spinlock.h in xv6 [15]

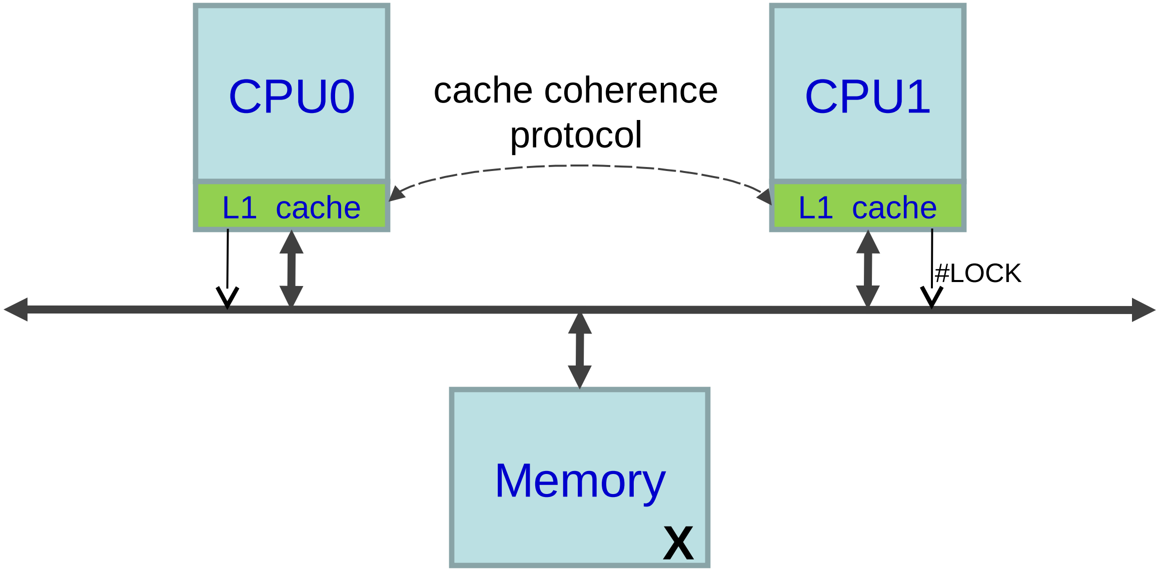
Issues with Spinlocks

xchg %eax, X

- No compiler optimizations should be allowed
  - Should not make X a register variable
    - Write the loop in assembly or use volatile
- Should not reorder **memory** loads and stores
    - Use serialized instructions (which forces instructions not to be reordered)
    - Luckly xchg is already implements serialization

More issues with Spinlocks

xchg %eax, X



- No caching of (X) possible. All xchg operations are bus transactions.
  - CPU asserts the LOCK, to inform that there is a ‘locked ‘ memory access
- acquire function in spinlock invokes xchg in a loop…each operation is a bus transaction …. huge performance hits

A better acquire

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}
```

```
void acquire(int *locked){
  reg = 1
  while(1)
      if(xchg(locked, reg) == 0)
          break;
}
```

Original.
Loop with xchg.
Bus transactions.
Huge overheads

```
void acquire(int *locked) {
  reg = 1;
  while (xchg(locked, reg) == 1)
        while (*locked == 1);
}
```

✓

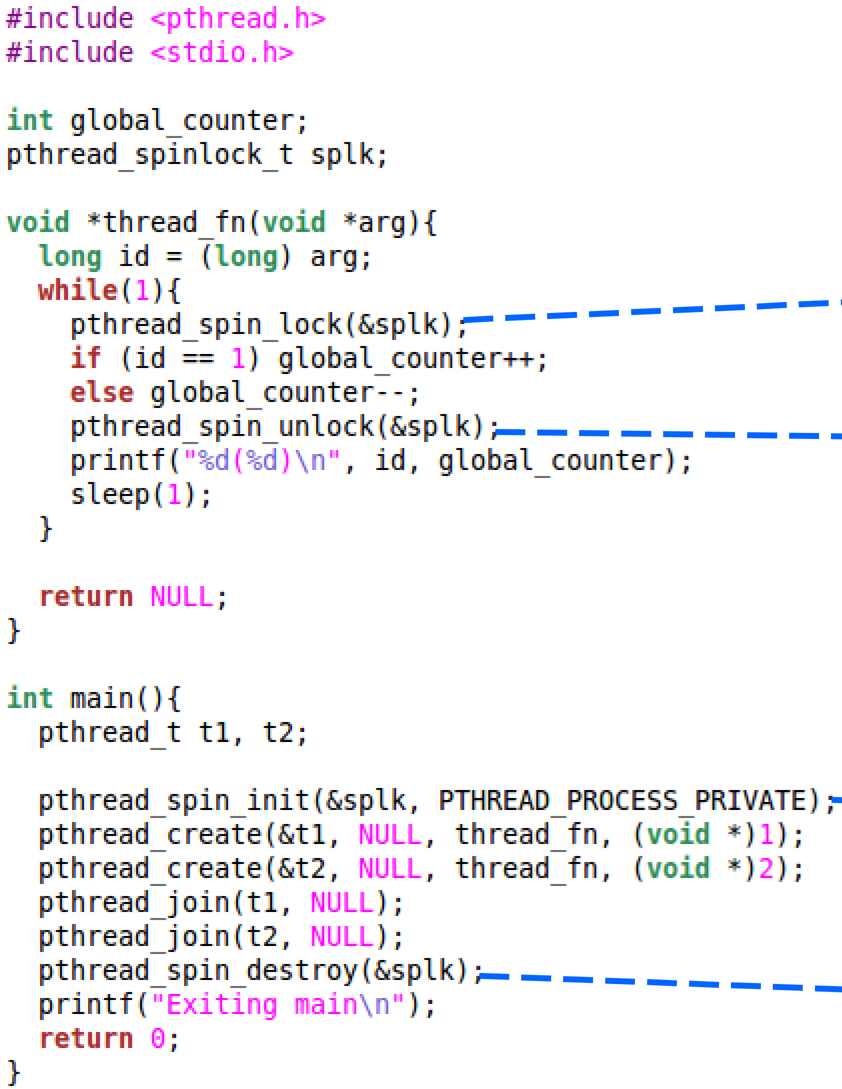
Better way
inner loop allows caching of locked. Access cache instead of memory.

Spinlocks
(when should it be used?)

- Characteristic : busy waiting
  - Useful for short critical sections, where much CPU time is not wasted waiting
    - eg. To increment a counter, access an array element, etc.

  - Not useful, when the period of wait is unpredictable or will take a long time
    - eg. Not good to read page from disk.
    - Use mutex instead (…mutex)

Spinlock in pthreads



```
#include <pthread.h>
#include <stdio.h>

int global_counter;
pthread_spinlock_t splk;

void *thread_fn(void *arg){
  long id = (long) arg;
  while(1){
    pthread_spin_lock(&splk);
    if (id == 1) global_counter++;
    else global_counter--;
    pthread_spin_unlock(&splk);
    printf("%d(%d)\n", id, global_counter);
    sleep(1);
  }

  return NULL;
}

int main(){
  pthread_t t1, t2;

  pthread_spin_init(&splk, PTHREAD_PROCESS_PRIVATE);
  pthread_create(&t1, NULL, thread_fn, (void *)1);
  pthread_create(&t2, NULL, thread_fn, (void *)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_spin_destroy(&splk);
  printf("Exiting main\n");
  return 0;
}
```

lock

unlock

create spinlock

destroy spinlock

Mutexes

- Can we do better than busy waiting?
  - If critical section is locked then yield CPU
    - Go to a SLEEP state
  - While unlocking, wake up sleeping process

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
       sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

Ref: wakeup(2864), sleep(2803)

Thundering Herd Problem

- A large number of processes wake up (almost simultaneously) when the event occurs.
  - All waiting processes wake up
  - Leading to several context switches
  - All processes go back to sleep except for one, which gets the critical section
    - Large number of context switches
    - Could lead to starvation

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
       sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

Thundering Herd Problem

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else{
      // add this process to Queue
       sleep();
    }
  }
}

void unlock(int *locked){
    locked = 0;
    // remove process P from queue
    wakeup(P)
}
```

pthread Mutex

- pthread_mutex_lock
- pthread_mutex_unlock

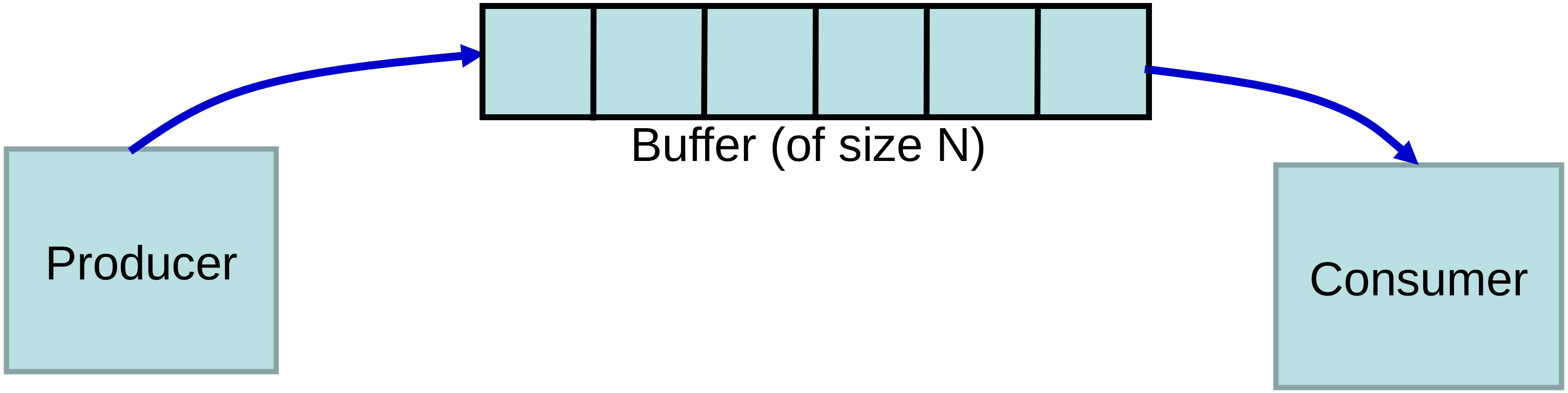
Locks and Priorities

- What happens when a high priority task requests a lock, while a low priority task is in the critical section
  - – Priority Inversion
  - – Possible solution
    - Priority Inheritance

Interesting Read : Mass Pathfinder
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html

Producer – Consumer Problems

- Also known as *Bounded buffer Problem*
- Producer produces and stores in buffer, Consumer consumes from buffer
- Trouble when
  - Producer produces, but buffer is full
  - Consumer consumes, but buffer is empty



Producer-Consumer Code

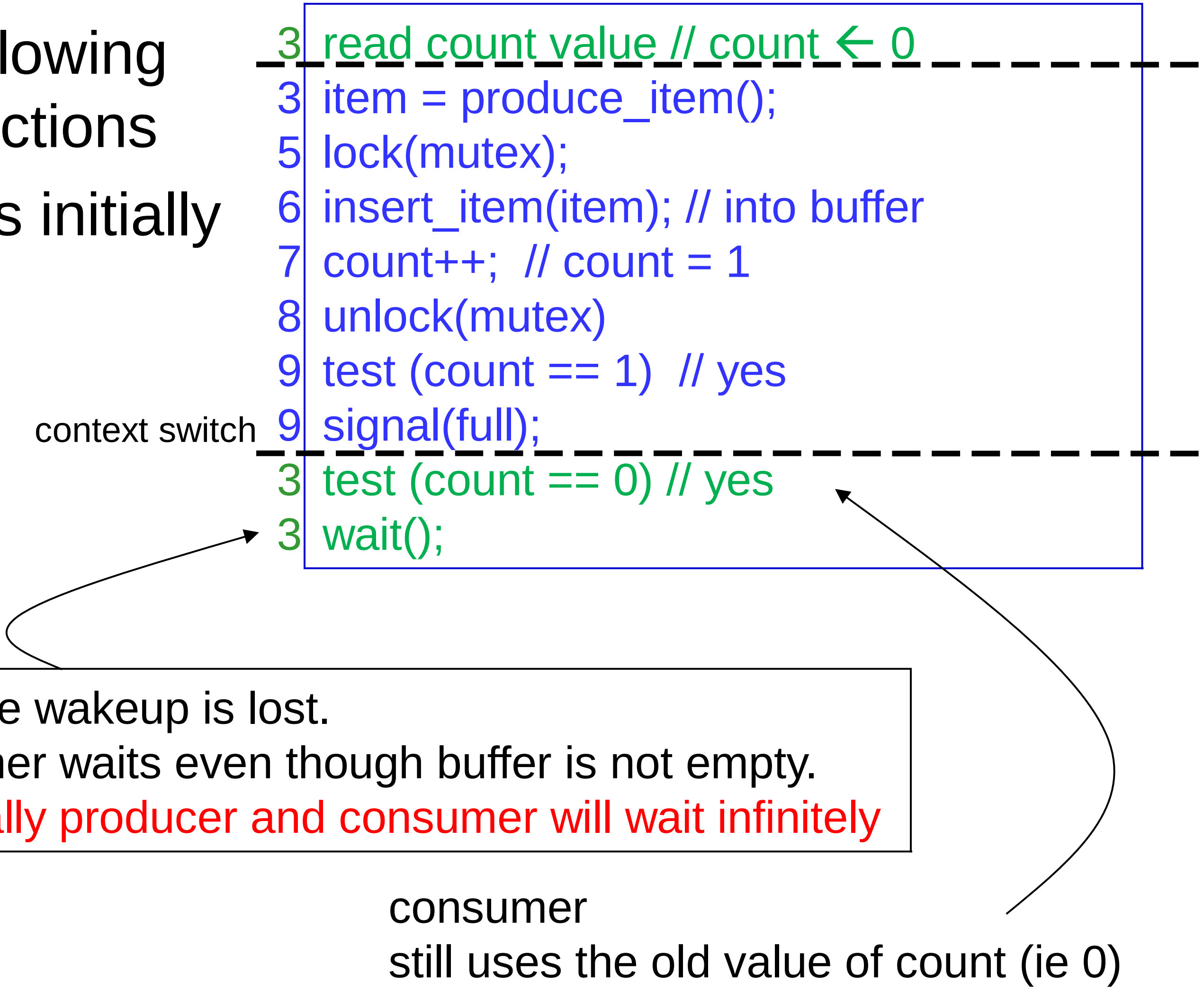
```
Buffer of size N
int count=0;
Mutex mutex, empty, full;
```

```
void producer(){
    while(TRUE){
        item = produce_item();
        if (count == N) sleep(empty);
        lock(mutex);
        insert_item(item); // into buffer
        count++;
        unlock(mutex);
        if (count == 1) wakeup(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        if (count == 0) sleep(full);
        lock(mutex);
        item = remove_item(); // from buffer
        count--;
        unlock(mutex);
        if (count == N-1) wakeup(empty);
        consume_item(item);
    }
}
```

Lost Wakeups

- Consider the following context of instructions
- Assume buffer is initially empty



```
3 read count value // count ← 0
3 item = produce_item();
5 lock(mutex);
6 insert_item(item); // into buffer
7 count++;  // count = 1
8 unlock(mutex)
9 test (count == 1)  // yes
9 signal(full);
3 test (count == 0) // yes
3 wait();
```

context switch

Note, the wakeup is lost.
Consumer waits even though buffer is not empty.
Eventually producer and consumer will wait infinitely

consumer
still uses the old value of count (ie 0)

Semaphores

- Proposed by Dijkstra in 1965
- Functions down and up must be atomic
- down also called **P** (Proberen Dutch for try)
- up also called **V** (Verhogen, Dutch form make higher)
- Can have different variants
  - Such as blocking, non-blocking
- If S is initially set to 1,
  - Blocking semaphore similar to a Mutex
  - Non-blocking semaphore similar to a spinlock

```
void down(int *S){
  while( *S <= 0);
  *S--;
}

void up(int *S){
   *S++;
}
```

Producer-Consumer with Semaphores

Buffer of size N
int count;

full = 0, empty = N

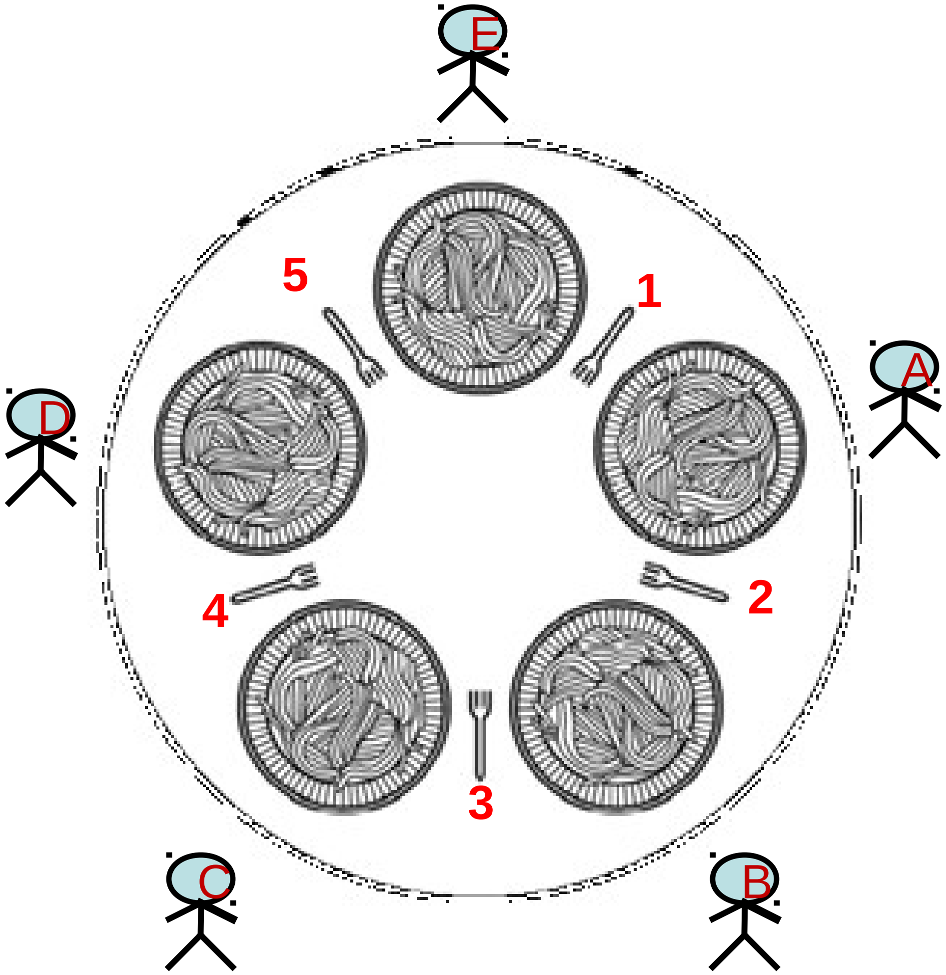
```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);
        wait(mutex);
        insert_item(item); // into buffer
        signal(mutex);
        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);
        wait(mutex);
        item = remove_item(); // from buffer
        signal(mutex);
        up(empty);
        consume_item(item);
    }
}
```

POSIX semaphores

- sem_init
- sem_wait
- sem_post
- sem_getvalue
- sem_destroy

Dining Philosophers Problem



- **Philosophers either think or eat**

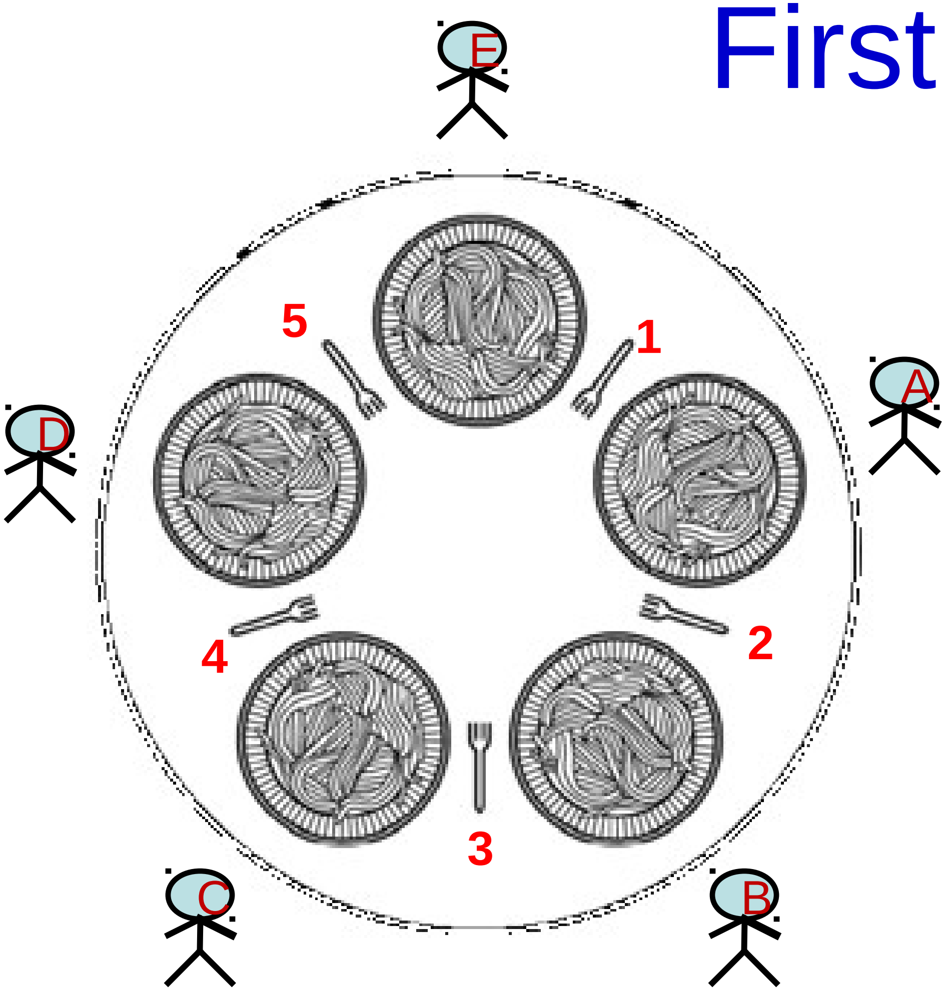
- To eat, a philosopher needs to hold both forks (the one on his left and the

  one on his right)

- If the philosopher is not eating, he is thinking.

- Problem Statement : Develop an algorithm where no philosopher starves.

First Try

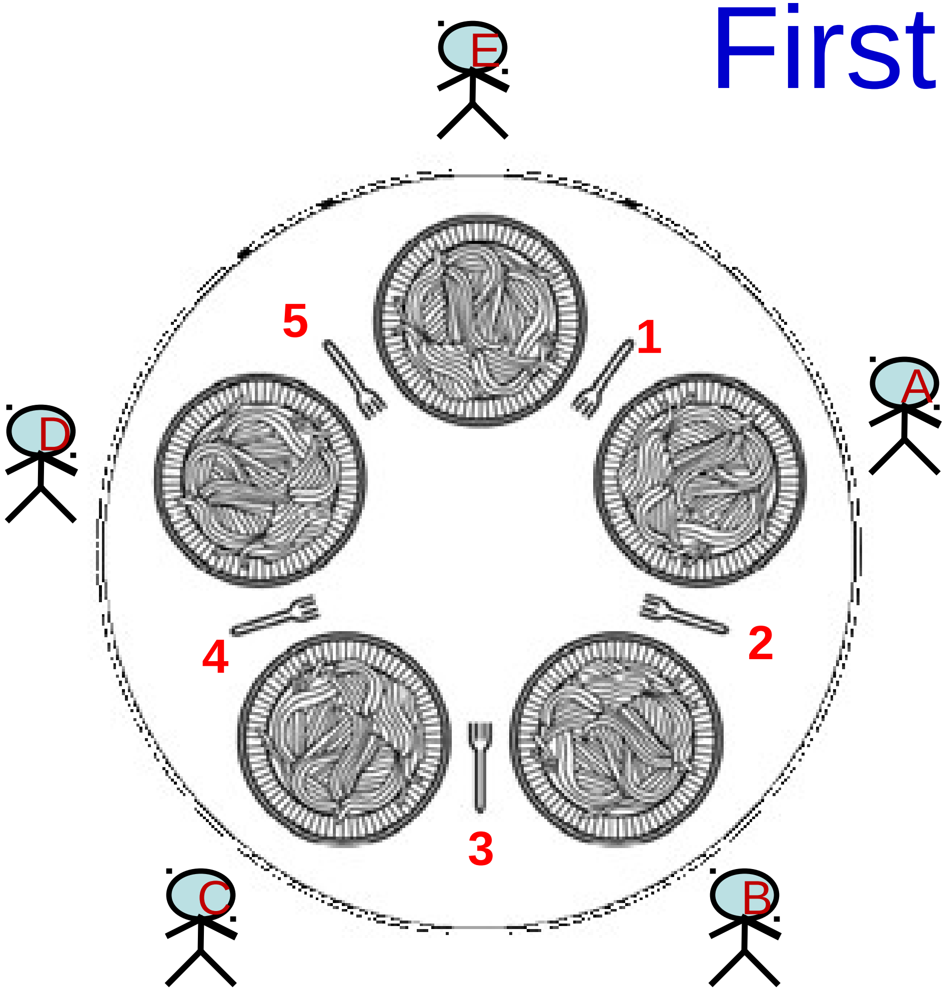


```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(i);
      take_fork((i + 1) % N);
      eat();
      put_fork(i);
      put_fork((i + 1) % N);
  }
}
```

What happens if only philosophers A and C are always given the priority?

B, D, and E starves… so scheme needs to be fair

First Try



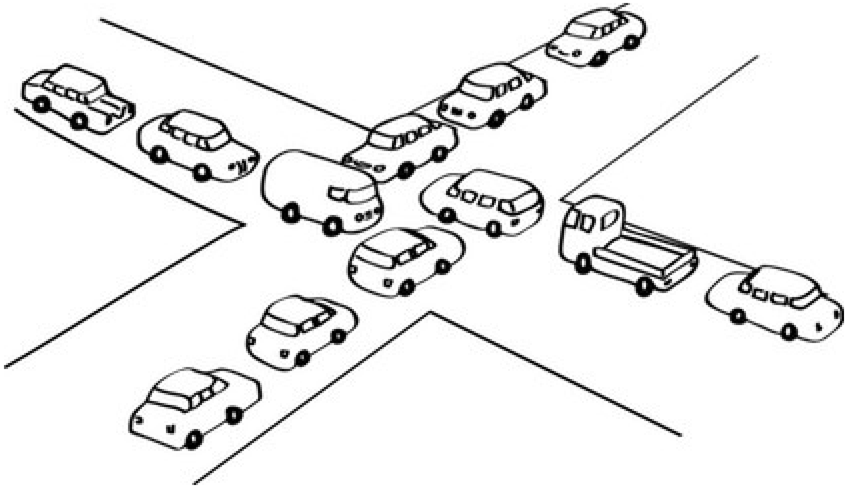
```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(i);
      take_fork((i + 1) % N);
      eat();
      put_fork(i);
      put_fork((i + 1) % N);
  }
}
```

What happens if all philosophers decide to pick up their left forks at the same time?

Possible starvation due to deadlock

Deadlocks

- A situation where programs continue to run indefinitely without making any progress
- Each program is waiting for an event that another process can cause



Second try

- **Take fork i, check if fork (i+1)%N is available**
- Imagine,
  - All philosophers start at the same time
  - Run simultaneously
  - And think for the same time
- This could lead to philosophers taking fork and putting it down continuously. a deadlock.

- A better alternative
  - Philosophers wait a random time before take_fork(i)
  - Less likelihood of deadlock.
  - Used in schemes such as Ethernet

```
#define N 5

void philosopher(int i){
  while(TRUE){
      think();
      take_fork(i);
      if (available((i+1)%N){
         take_fork((i + 1) % N);
         eat();
      }else{
          put_fork(i);
     }
}
```

Solution using Mutex

- Protect critical sections with a mutex
- Prevents deadlock
- But has performance issues
  - Only one philosopher can eat at a time

```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      wait(mutex);
      take_fork(i);
      take_fork((i + 1) % N);
      eat();
      put_fork(i);
      put_fork((i + 1) % N);
      signal(mutex);
  }
}
```

Solution to Dining Philosophers

Uses N semaphores (s[0], s[1], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

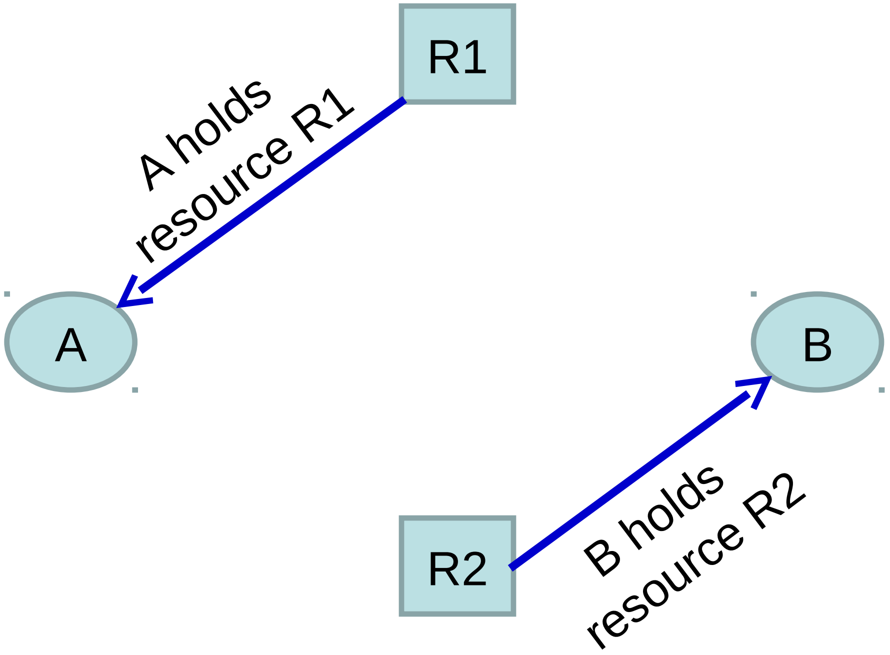
```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
     lock(mutex);
     state[i] = HUNGRY;
     test(i);
     unlock(mutex);
     down(s[i]);
}
```

```
void put_forks(int i){
     lock(mutex);
     state[i] = THINKING;
     test(LEFT);
     test(RIGHT)
     unlock(mutex);
}
```

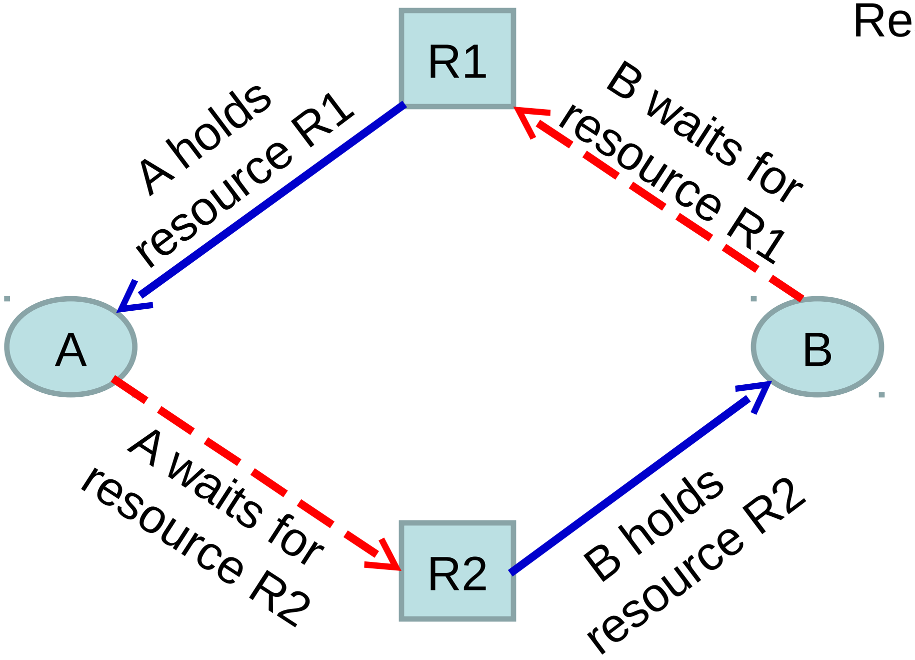
```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

Deadlocks



Consider this situation:

Deadlocks



Resource Allocation Graph

A Deadlock Arises:

Deadlock : 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.

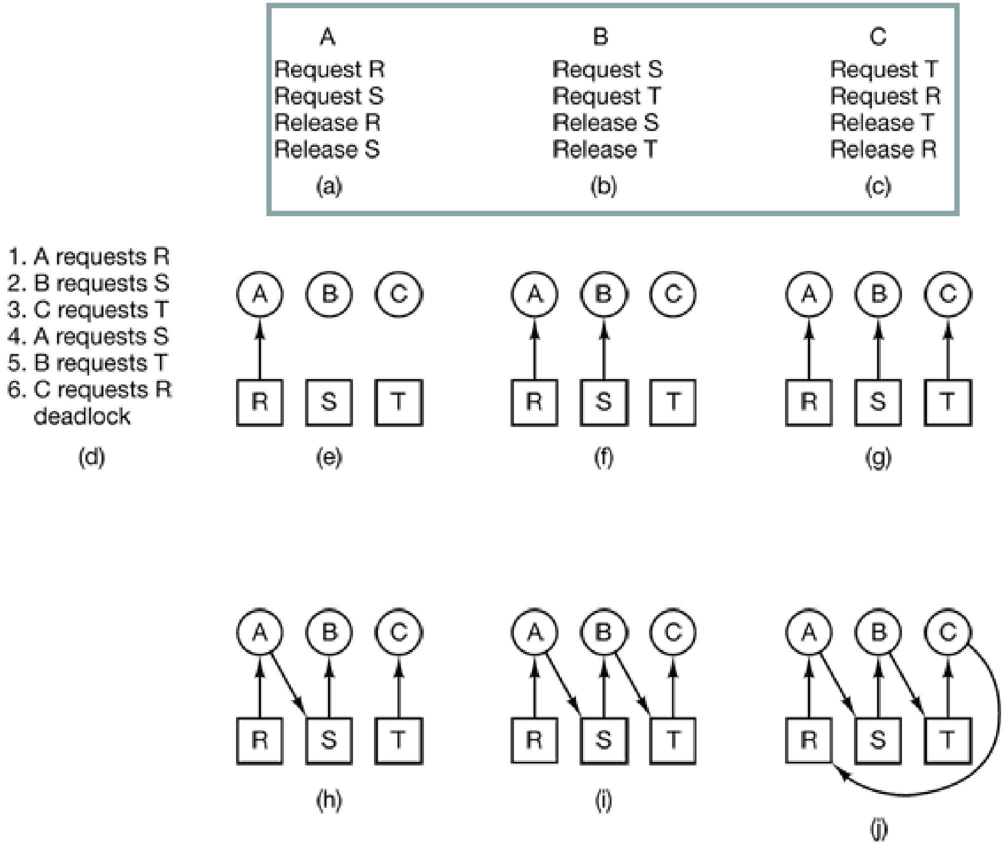
Conditions for Resource Deadlocks

1. **Mutual Exclusion**
    - Each resource is either available or currently assigned to exactly one process
2. **Hold and wait**
    - A process holding a resource, can request another resource
3. **No preemption**
    - Resources previously granted cannot be forcibly taken away from a process
4. **Circular wait**
    - There must be a circular chain of two or more processes, each of which is waiting for a resouce held by the next member of the chain

All four of these conditions must be present for a resource deadlock to occur!!

Deadlocks :
(A Chanced Event)

- Ordering of resource requests and allocations are probabilistic, thus deadlock occurrence is also probabilistic

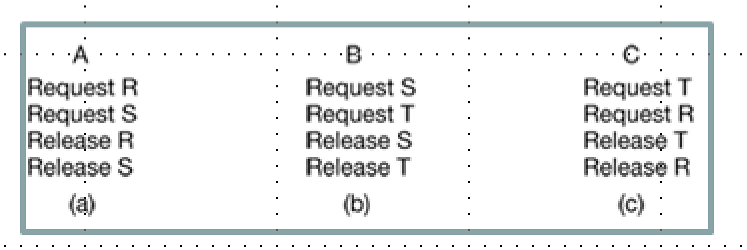


A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R
(a)	(b)	(c)

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)

Deadlock occurs



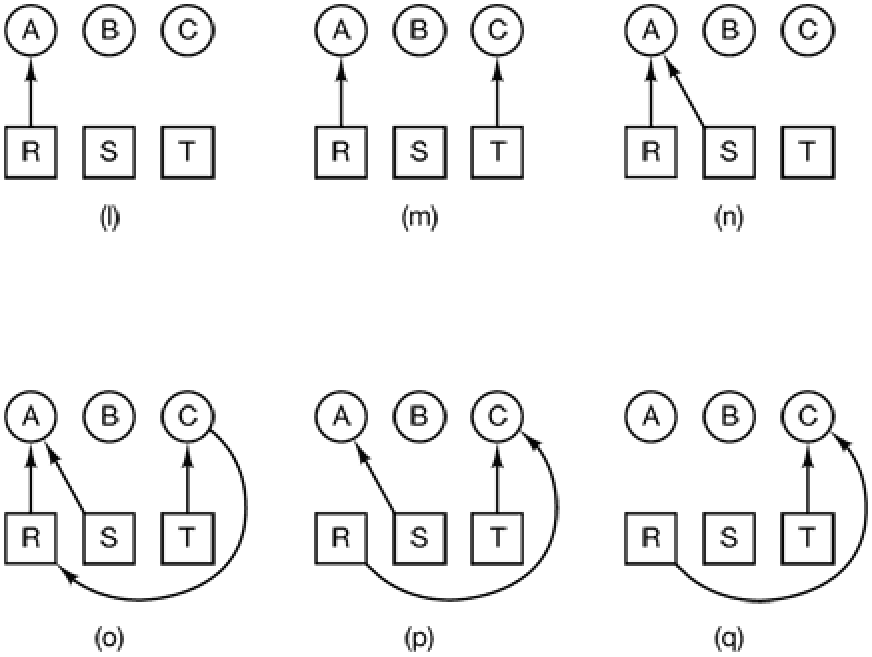
A	B	C
Request R Request S Release R Release S	Request S Request T Release S Release T	Request T Request R Release T Release R
(a)	(b)	(c)

No dead lock occurrence
(B can be granted S
after step q)

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)

Should Deadlocks be handled?

- Preventing / detecting deadlocks could be tedious
- Can we live without detecting / preventing deadlocks?
  - What is the probability of occurrence?
  - What are the consequences of a deadlock? (How critical is a deadlock?)

Handling Deadlocks

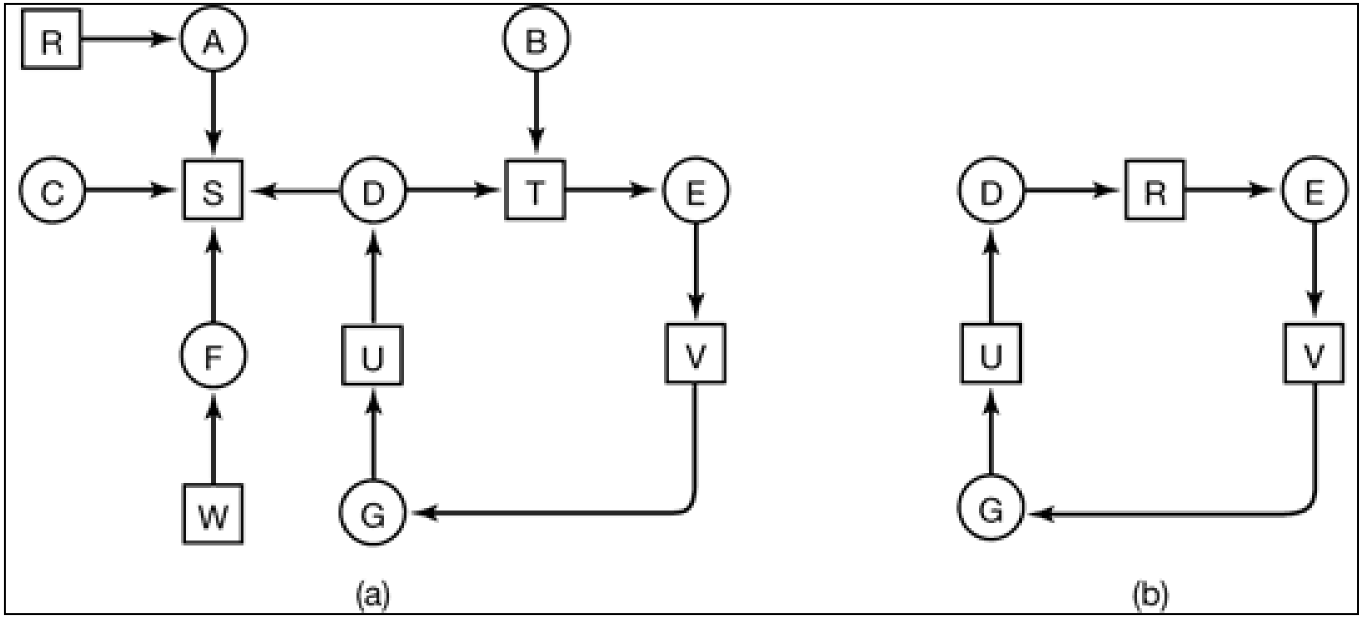
- Detection and Recovery
- Avoidance
- Prevention

Deadlock detection

- How can an OS detect when there is a deadlock?
- OS needs to keep track of
  - Current resource allocation
    - Which process has which resource
  - Current request allocation
    - Which process is waiting for which resource
- Use this informaiton to detect deadlocks

Deadlock Detection

- Deadlock detection with **one resource of each type**
- Find cycles in resource graph



Deadlock Detection

- Deadlock detection with multiple resources of each type

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1) \end{array}$$

Existing Resource Vector

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 2 & 1 & 0 & 0) \end{array}$$

Resources Available

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Current allocation matrix

$$\begin{array}{l} P_1 \\ P_2 \\ P_3 \end{array} \quad C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Current Allocation Matrix

Who has what!!

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Request Matrix

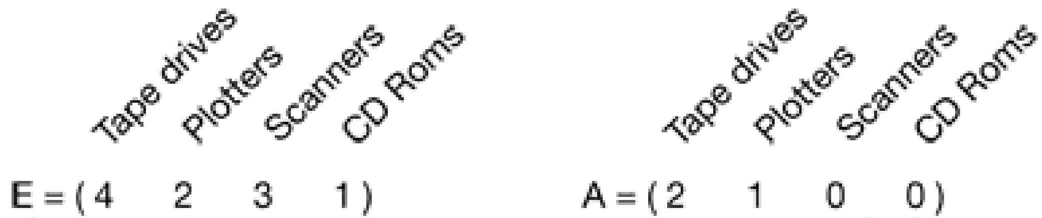
Who is waiting for what!!

Process P_i holds C_i resources and requests R_i resources, where $i = 1$ to 3

Goal is to check if there is any sequence of allocations by which all current requests can be met. If so, there is no deadlock.

Deadlock Detection

- Deadlock detection with multiple resources of each type



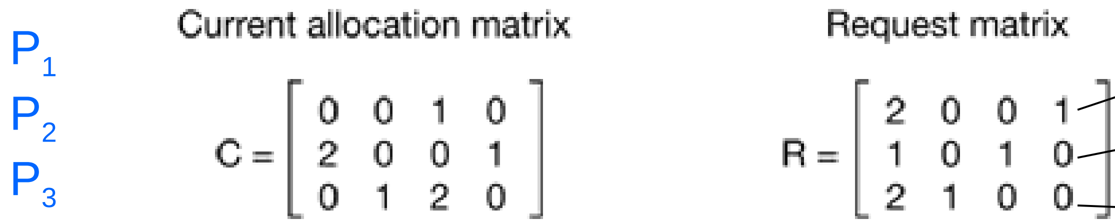
$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

Existing Resource Vector

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \end{array}$$

Resources Available

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$



Current allocation matrix

$$\begin{array}{cc} P_1 & \\ P_2 & C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \\ P_3 & \end{array}$$

Current Allocation Matrix

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Request Matrix

P_1 cannot be satisfied

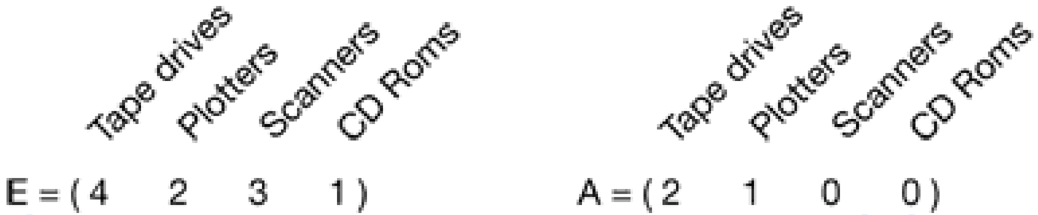
P_2 cannot be satisfied

P_3 can be satisfied

Process P_i holds C_i resources and requests R_i resources, where $i = 1$ to 3

Deadlock Detection

- Deadlock detection with multiple resources of each type

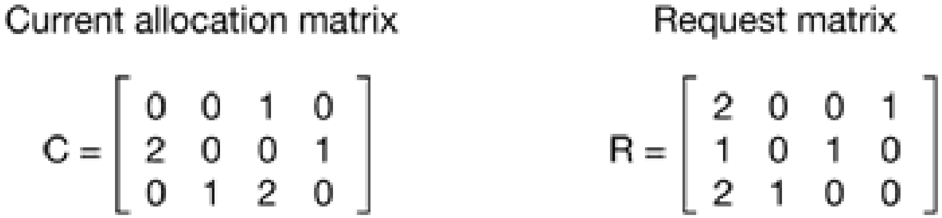


$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array}$$

Existing Resource Vector

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (2 & 1 & 0 & 0) \end{array}$$

Resources Available



Current allocation matrix

$$\begin{array}{c} P_1 \\ P_2 \\ P_3 \end{array} \quad C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Current Allocation Matrix

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Request Matrix

P_3 runs and its allocation is (2, 2, 2, 0)

On completion it returns the available resources are A = (4 2 2 1)

Either P_1 or P_2 can now run.

NO Deadlock!!!

Deadlock Detection

- Deadlock detection with multiple resources of each type

$$\begin{array}{ccccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1\,) \end{array}$$

Existing Resource Vector

$$\begin{array}{ccccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 2 & 1 & 0 & 0\,) \end{array}$$

Resources Available

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Current allocation matrix

$$\begin{array}{cc} \begin{array}{c} P_1 \\ P_2 \\ P_3 \end{array} & C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \end{array}$$

Current Allocation Matrix

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

Request Matrix

P_1 cannot be satisfied

P_2 cannot be satisfied

P_3 cannot be satisfied

deadlock

Process P_i holds C_i resources and requests R_i resources, where $i = 1$ to 3

Deadlock detected as none of the requests can be satisfied

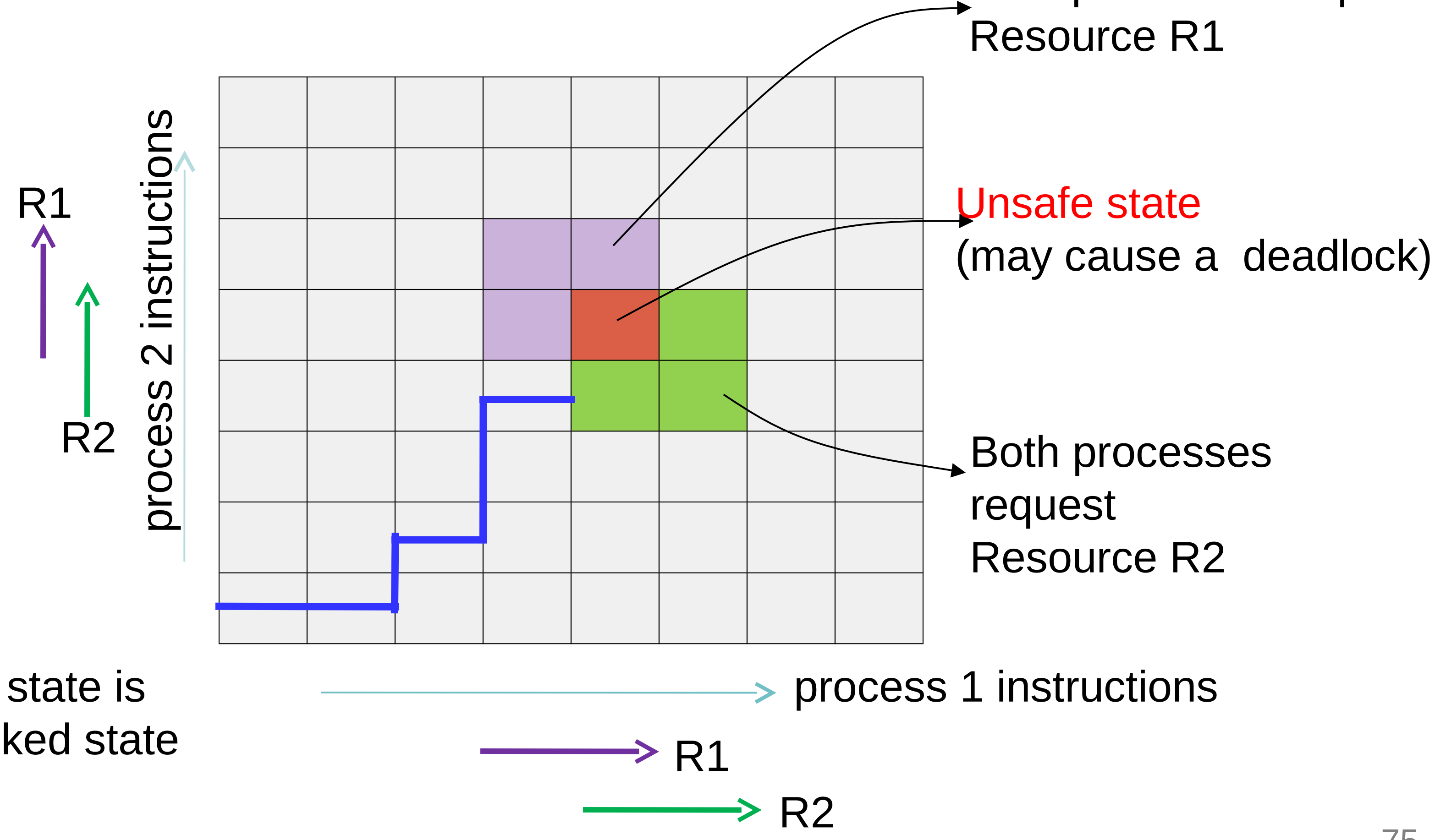
Deadlock Recovery

What should the OS do when it detects a deadlock?

- Raise an alarm
  - Tell users and administrator
- Preemption
  - Take away a resource temporarily (frequently not possible)
- Rollback
  - Checkpoint states and then rollback
- Kill low priority process
  - Keep killing processes until deadlock is broken
  - (or reset the entire system)

Deadlock Avoidance

- System decides in advance if allocating a resource to a process will lead to a deadlock



Both processes request Resource R1

Unsafe state
(may cause a deadlock)

Both processes request Resource R2

Note: unsafe state is not a deadlocked state

Deadlock Avoidance

Is there an algorithm that can always avoid deadlocks by conservatively make the right choice.

- Ensures system never reaches an unsafe state

- Safe state : A state is said to be safe, if there is some scheduling order in which every process can run to completion even if all of them suddenly requests their maximum number of resources immediately
- An unsafe state does not have to lead to a deadlock; *it* ***could*** *lead to a deadlock*

Example with a Banker

- Consider a banker with 4 clients (P_1, P_2, P_3, P_4).
  - Each client has certain credit limits (totaling 20 units)
  - The banker knows that max credits will not be used at once, so he keeps only 10 units

	Has	**Max**
A	3	9
B	2	4
C	2	7

Total : 10 units

free : 3 units

  - Clients declare maximum credits in advance. The banker can allocate credits provided no unsafe state is reached.

Safe State

	Has	Max
A	3	9
B	2	4
C	2	7

free : 3 units

Allocate 2 units to B

	Has	Max
A	3	9
B	4	4
C	2	7

free : 1 units

B completes

	Has	Max
A	3	9
B	0	-
C	2	7

free : 5 units

Allocate 5 to C

	Has	Max
A	3	9
B	0	-
C	7	7

free : 0 units

C completes

	Has	Max
A	3	9
B	0	-
C	0	-

free : 7 units

Allocate 6 units to A

	Has	Max
A	9	9
B	0	-
C	0	-

free : 0 units

This is a safe state because there is some scheduling order in which every process executes

Unsafe State

	Has	Max
A	4	9
B	2	4
C	2	7

free : 2 units

Allocate 2 units to B

	Has	Max
A	4	9
B	4	4
C	2	7

free : 0 units

B completes

	Has	Max
A	4	9
B	0	-
C	2	7

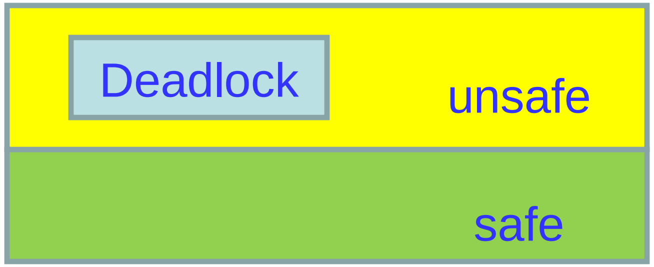
free : 4 units

This is an unsafe state because there exists NO scheduling order in which every process executes

Banker's Algorithm (with a single resource)

When a request occurs

- If(**is_system_in_a_safe_state**)
    - Grant request
- else
    - postpone until later



Deadlock

unsafe

safe

Please read Banker's Algorithm with multiple resources from Modern Operating Systems, Tanenbaum

Deadlock Prevention

- Deadlock avoidance not practical, need to know maximum requests of a process
- Deadlock prevention
  - Prevent at-least one of the 4 conditions
  1. **Mutual Exclusion**
  2. **Hold and wait**
  3. **No preemption**
  4. **Circular wait**

Prevention

1. **Preventing Mutual Exclusion**
   - Not feasible in practice
   - But OS can ensure that resources are optimally allocated
2. **Hold and wait**
   - One way is to achieve this is to require all processes to request resources before starting execution
     - May not lead to optimal usage
     - May not be feasible to know resource requirements
3. **No preemption**
   - Pre-empt the resources, such as by virtualization of resources (eg. Printer spools)
4. **Circular wait**
   - One way, process holding a resource cannot hold a resource and request for another one
   - Ordering requests in a sequential / hierarchical order.

Hierarchical Ordering of Resources

- Group resources into levels

  (i.e. prioritize resources numerically)
- A process may only request resources at higher levels than any resource it currently holds
- Resource may be released in any order
- eg.
  - Semaphore s1, s2, s3 (with priorities in increasing order)

    down(S1); down(S2); down(S3) ; → allowed

    down(S1); down(S3); down(S2); →not allowed