Systems Software/Programming

Concurrent Programming

Motivation Concurrent Programming

- Improve the application performance by running multiple tasks simultaneously (Divide the task into n smaller pieces, using n processors run it n times faster) → mainly used in computationally intensive processes
  - addition of 100,000 elements available in memory
  - convert RGB image to grayscale by applying 0.2126R+0.7152G+0.0722B to each pixel.
- To cop with independent slower devices → mainly used in I/O bound processes e.g. running multiple database queries since database is mostly disk I/O intensive application
  - Do not wait for slower devices such as disk, network etc. instead perform other tasks
    - During I/O perform computation
    - During continuous visualization, handle key strokes and I/O e.g. video games
    - While listening to network, perform other operations e.g. listening to multiple sockets at the same time
    - Concurrent I/O, concurrent transfers e.g. Web browsers

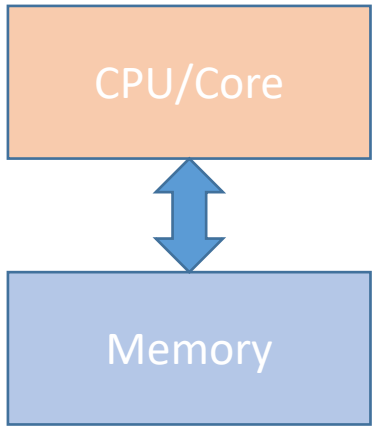
Sequential versus Concurrent Programming

- Sequential Programming : Multiple computational tasks are performed in sequential order
- Concurrent Programming : Multiple computational tasks are performed simultaneously
- Concurrent tasks can be implemented using
  - Multiple processes using parent-child model using fork() that we studied earlier OR
  - Multiple threads within the same process
- Execution of concurrent tasks can use:
  - Single processor (core) : multi-process or multithreaded programming where multiple processes or threads running concurrently
  - Several processors (cores) part of the same system i.e. in close proximity : parallel computing where multiple threads run in parallel
  - Several processors (cores) distributed across a network i.e. part of different systems : distributed computing

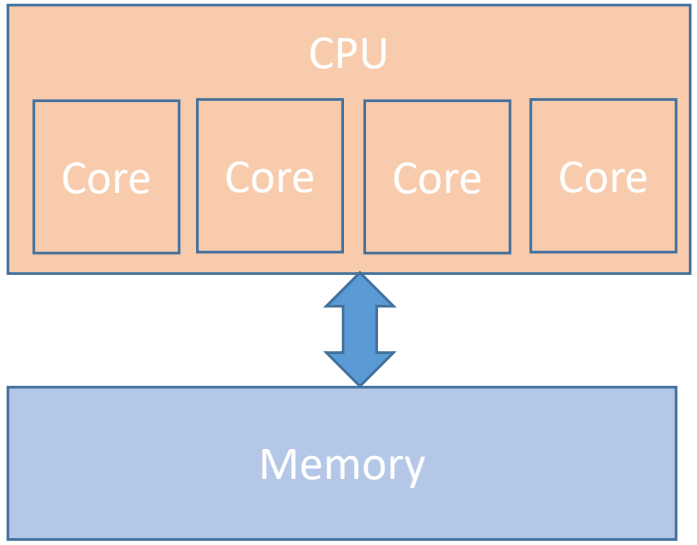
Interaction between Concurrent Tasks

- multi-process or multithreaded programming and parallel computing is meant for single systems → Shared memory architecture
  - All tasks has access to the same physical memory
  - Commination between tasks is done by changing content of the shared memory
- Distributed computing is meant for multiple systems connected through network → Distributed memory architecture
  - No sharing of physical memory
  - Tasks must communicate by exchanging messages

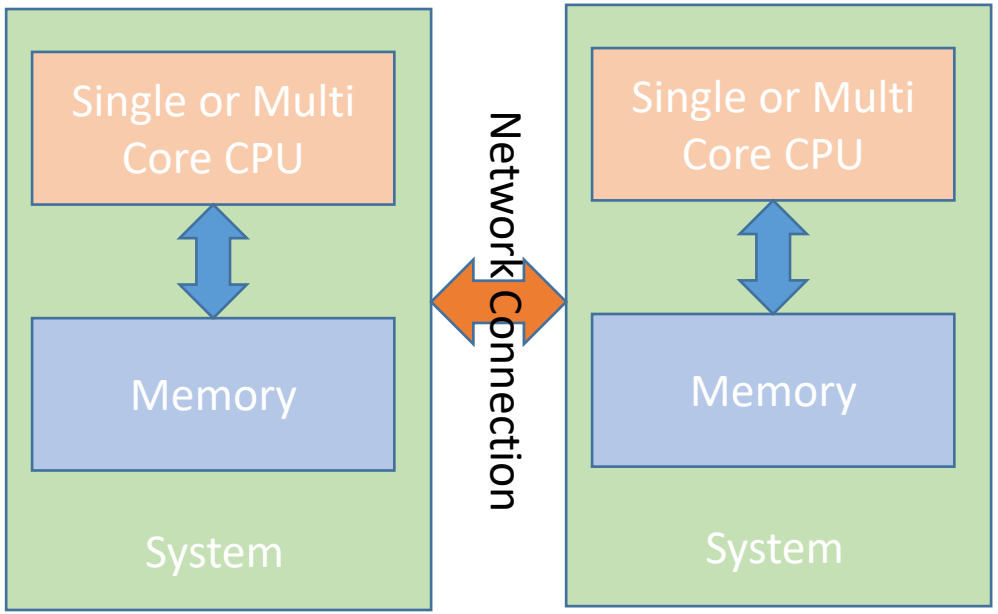
Shared Memory vs Distributed Memory Architectures



Single CPU/Core used for Concurrent Computing



Multi Core Shared Memory System used for Concurrent and Parallel Computing



Distributed Memory System used for Distributed Computing

Serial versus Parallel Tasks (Divide and Conquer)

x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8

Step1: sum=x1 + x2

Step2: sum=sum + x3

Step3: sum=sum + x4

Step4: sum=sum + x5

Step5: sum=sum + x6

Step6: sum=sum + x7

Step7: sum=sum + x8

How may total steps? 7 ~ 8

x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8

With 4 processors divide tasks in 4

Step1: parallelism = 4

a=x1 + x2 b=x3 + x4
c=x5 + x6 d=x7 + x8

Now using 2 processors calculate

Step2: parallelism = 2

e=a+b f=c+d

Now using 1 processor calculate

Step3: parallelism = 1

sum=e+f

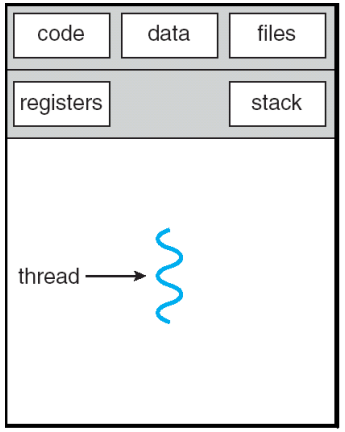
How many total steps? 3 = log(8)

Gain from Parallelism

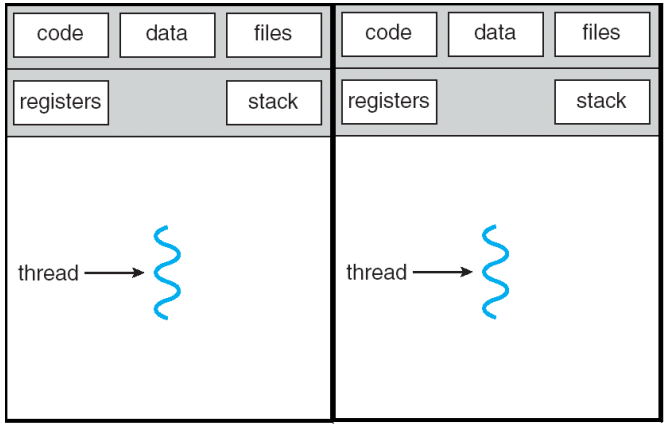
- In theory,
  - Diving program into p small parts and running them in parallel on p processors should result in p times speed up (means it should run p time faster), because
    - Time taken by parallel version algorithm (Tp) = Time taken by serial algorithms (Ts) / Number of processors (p)
    - Speedup (s) = Ts / Tp → this should be equal to number of processors (p)

- In practice, this is the case due to
  - Inter-task dependencies
    - In addition example Ts=n, Tp=log(n), therefore speedup(s)=n/log(n), for n=8; Ts=,Tp=log(8)=3 therefore speedup(s)=8/3=2.67. Ideally it should have been equal to processors p=4.
  - Communication Costs
    - In case data is very large which need to send across network, it will incur communication cost e.g. reading 100,000 elements from use in one main process and then sending partial results of additions to other processors

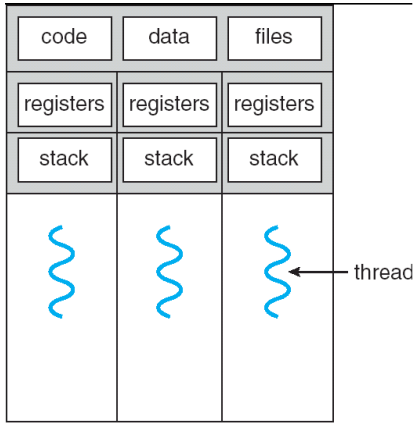
Memory Map of Multi-process vs Multithreading



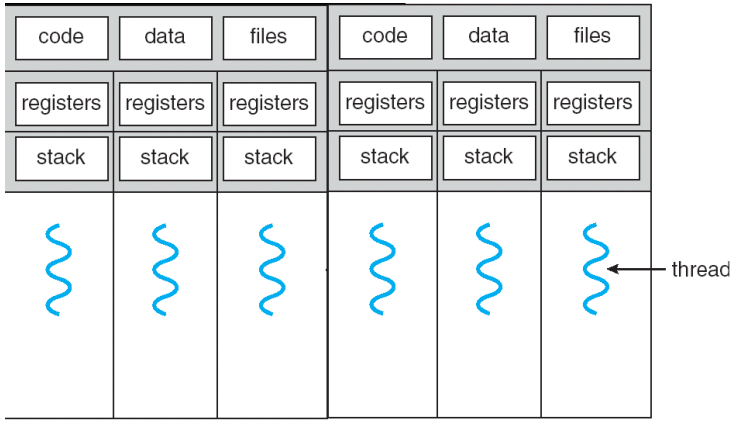
Single Process Single Thread



Multi-Process Single Thread



Single Process Multithreaded



Multi-Process Multithreaded

Multi-process Model vs Multithreaded Model

- Process Spawning
- Process creation involve following actions:
- Setting up process control block (PCB)
- Allocation of memory address space
- Loading program into allocated memory address space
- Passing PCB to scheduler to queue up the process to run

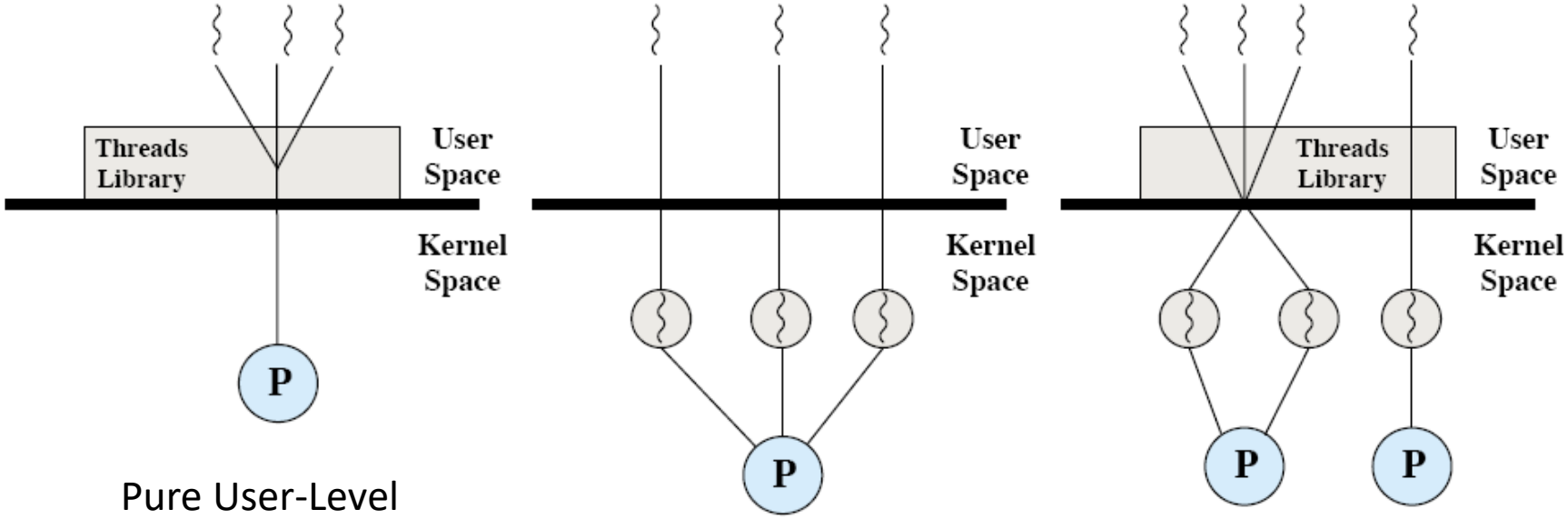
- Thread Spawning
- Threads are created within and belonging to processes
- All the threads created within one process share the resources of the process including the address space
- Scheduling is performed on a per-thread basis.
- The thread model is a finer grain scheduling model than the process model
- Threads have a similar lifecycle as the processes and will be managed mainly in the same way as processes are

Why use multi-threaded model over multi-process model?

- A common terminology:
  - Heavyweight Process = Process
  - Lightweight Process = Thread
- Advantages (Thread vs. Process):
  - Much quicker to create a thread than a process
    - spawning a new thread only involves allocating a new stack and a new CPU state block
  - Much quicker to switch between threads than to switch between processes
  - Threads share data easily
- Disadvantages (Thread vs. Process):
  - Processes are more flexible – because they use different memory map
  - No security between threads: One thread can stomp on another thread's data
  - For threads which are supported by user thread package instead of the kernel:
    - If one thread blocks, all threads in task block.

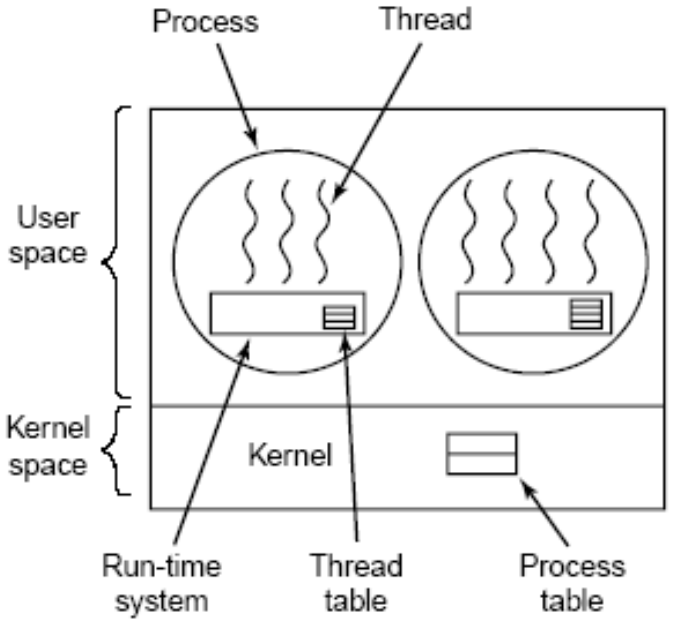
Thread Implementation

- Two broad categories of thread implementation
  - User-Level Threads (ULT)
  - Kernel-Level Threads (KLT)



Thread Implementation

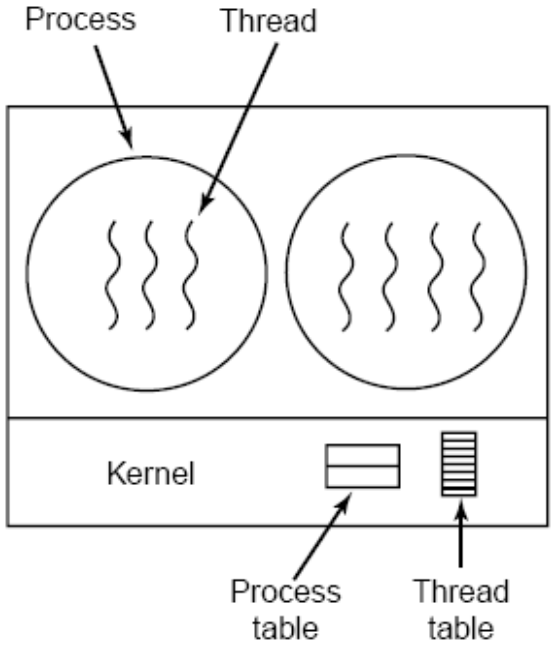
- User-Level Threads (ULT)
  - Kernel is not aware of existence of threads, it knows only processes with one thread of execution
  - Each user process manages its own private thread table
  - Advantages:
    - light thread switching: does not need kernel mode privileges
    - cross-platform: ULT can run on any underlying O/S
  - Disadvantage:
    - if a thread blocks, the entire process is blocked, including all other threads in it



User-Level Thread Package

Thread Implementation

- Kernel-Level Thread (KLT)
  - the kernel knows about and manages the threads: creating and destroying threads are system calls
  - Advantage:
    - fine-grain scheduling, done on a thread basis
    - if a thread blocks, another one can be scheduled without blocking the whole process
  - Disadvantage:
    - heavy thread switching involving mode switch



Kernel-Level Thread Package

POSIX Threads (pthreads) Library

- OS Support: FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Android and Solaris. DR-DOS and Microsoft Windows implementation also exists
- Library : libpthread
- Header file : pthread.h
- More then 100 threads functions categorized in to 4 category
  - Thread Management
  - Mutexes
  - Condition Variables
  - Synchronization between Threads using read/write locks and barriers
- To compile with gcc you need to use –pthread option of gcc to link libpthread

Thread Identification

ConcurrentProgramming\pthread_self_example.c

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns thread ID of the calling thread

Note: Each process has at least a single thread running by default even if you have not created thread explicitly

Thread Creation

#include <pthread.h>

int pthread_create(pthread_t *tidp, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

Returns 0 on success and error number in case of failure

Parameters:

- tidp : pointer to thread ID variable which will be set with thread ID for newly created thread
- attr : specific attributes to control the thread (default value to pass is NULL)
- start_routine : pointer to function which will be executed upon creation of a new thread
- arg: pointer to argument(s) to be passed to start_routine function. Note if more then one parameter need to be passed than you need to use structure instead of primitive type

Thread Creation Examples

- Create thread with function with no parameters.

ConcurrentProgramming\thread_example_no_param.c

- Create thread with function with only one parameter with primitive data type.

ConcurrentProgramming\thread_example_one_param.c

- Create thread with function with multiple parameters with primitive type or user-defined type.

ConcurrentProgramming\thread_example_multiple_params.c

Thread Termination

- If thread calls exit() system call, complete process will be terminated which is generally not desirable because the goal is to terminate a thread and not the process, hence we use pthread_exit() as below

#include <pthread.h>

void pthread_exit(void * retval);

Does not return anything

Different between using exit(), return or pthread_exit()

- exit() – exit the complete process so if you call from one of the thread function, it will end the process without waiting for main or any other thread in the process to continue → process ends
- ConcurrentProgramming\thread_example1.c

$ ./thread_example1.out

Main thread 521172736 is starting

child thread id 513017600 is starting

child thread id 513017600 is calling exit

$

Note: Message “Main thread 521172736 is finished” is never displayed because parent also have terminated when child called exit()

Different between using exit(), return or pthread_exit()

- return() – when called from a thread then it will not wait for any of child thread to complete → parent and all the child threads ends

ConcurrentProgramming\thread_example2.c

$ ./thread_example2.out

Main thread 1203812096 is starting

child thread id 1195656960 is starting

Main thread 1203812096 is finished

$

Note: Message “child thread id 1195656960 is calling exit” is never displayed because child is terminated by parent calling return

Different between using exit(), return or pthread_exit()

- pthread_exit() – when called from a thread then it will only end that thread but all other threads in the same process or child threads can continue → this thread ends but all the child threads or any other threads in the same process continues

ConcurrentProgramming\thread_example3.c

$ ./thread_example3.out

Main thread 1709131520 is starting

child thread id 1700976384 is starting

Main thread 1709131520 is finished but let the child thread continue

child thread id 1700976384 is calling exit

$

Note: pthread_exit() is called by parent/main thread hence it has terminated but child thread continues to execute until it calls exit()

Thread Synchronization

#include <pthread.h>

int pthread_join(pthread_t tid, void **retval);

thread calling pthread_join waits for target thread with tid to terminate. Target thread is terminated when it calls pthread_exit()

Returns 0 on success and error number in case of failure

Parameters:

tid : thread ID of target thread for which calling thread is waiting to terminate

retval : when target thread called pthread_exit(), exit status of the same is returned in retval

Complete Thread Example with Synchronization

ConcurrentProgramming\thread_sync_example.c

```
$ ./thread_sync_example.out
Main thread is 2739095296
child thread 2730940160 is created
child thread 2722547456 is created
main thread 2739095296 will wait for child thread 2730940160
child thread 2730940160 exiting
child thread 2722547456 exiting
child thread 2730940160 exit code 1
main thread 2739095296 will wait for child thread 2722547456
child thread 2722547456 exit code 2
Main thread 2739095296 exiting
$
```

Thread Co-Operation

ConcurrentProgramming\thread_coop_example.c

Multiple runs may produce different results

```
$ ./thread_coop_example.out
counter = 0
counter = 2
counter = 3
counter = 4
counter = 5
```

```
$ ./thread_coop_example.out
counter = 1
counter = 2
counter = 3
counter = 3
counter = 4
$
```

Thread Synchronization

- One thread can request another thread belonging to the same process to terminate by calling pthread_cancel()

#include <pthread.h>

int pthread_cancel(pthread_t tid);

Returns 0 on success and error number in case of failure

Parameters:

tid : thread ID of target thread which is requested to be cancelled

Note: Behavior of the target thread will be the same as if it had called pthread_exit()

Process vs Thread System Calls

Process Primitive	Thread Primitive	Description
fork	pthread_create	Create a new flow of control
exit	pthread_exit	Exit from an existing flow of control
waitpid	pthread_join	Get exit status from flow of control
atexit	pthread_cancel_push	Register function to be called at exit from flow of control
getpid	pthread_self	Get ID of flow of control
abort	pthread_cancel	Request abnormal termination of flow of control

Concurrency Issues

- Most common way of communication between multiple processes or threads on the same system is using shared variables in shared memory
- Let's say two tasks are running a instruction x = x + 1. if processes are different and not sharing memory the result produces is the same. But what happens when x is a shared variable.
- Consider x = x + 1 is a non-atomic operation (since memory is non-atomic) and two process runs below steps:
  - Read value of x from memory to register (Reg <- x)
  - Increment value of x in register (Reg=Reg+1)
  - Write back the updated value in memory (Reg -> x)

Concurrency Issue

- Consider x = 8 initially and two concurrent Threads execute x = x+1

T1	T2	Output
Reg <- x		Reg=8
Reg=Reg+1		Reg=9
Reg -> x		X=9
	Reg <- x	Reg=9
	Reg=Reg+1	Reg=10
	Reg -> x	x=10

Output from serial execution of T1 and T2 which should be generated by concurrent processes as well

T1	T2	Output
Reg <- x		Reg=8
Reg=Reg+1		Reg=9
	Reg <- x	Reg=8
	Reg=Reg+1	Reg=9
Reg -> x		x=9
	Reg -> x	x=9

Output from concurrent (interleaved) execution of T1 and T2

T1	T2	Output
Reg <- x		Reg=8
	Reg <- x	Reg=8
Reg=Reg+1		Reg=9
Reg -> x		x=9
	Reg=Reg+1	Reg=9
	Reg -> x	x=9

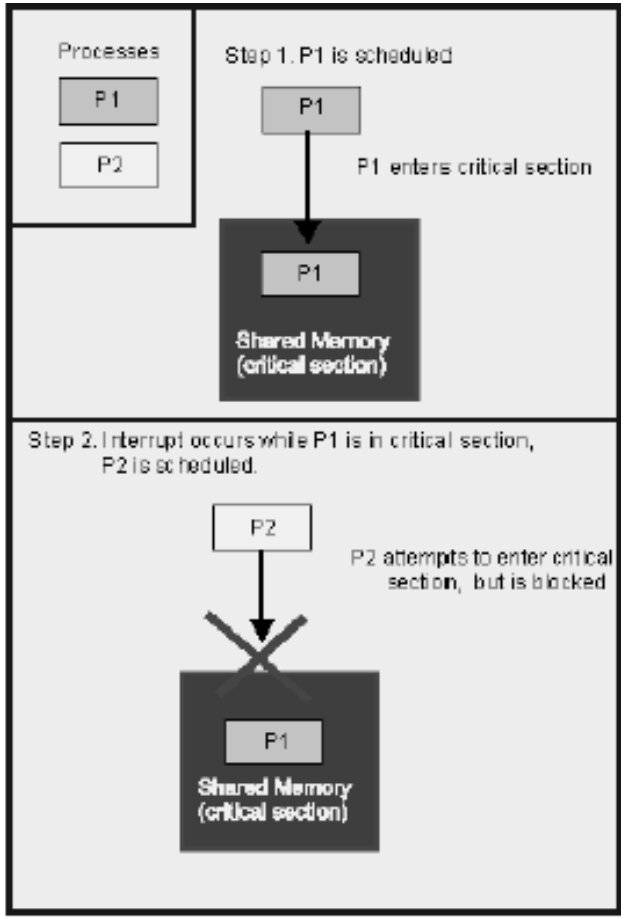
Output from concurrent (interleaved) execution of T1 and T2

Thread Synchronization

- Mechanism that allows programmer to control relative order of operation occurrence in different threads or processes
- How thread synchronization works:
  - Programmer identifies critical section in the code
  - Implements mutual exclusion to ensure that critical section is mutually exclusive i.e. atomic

Critical Section and Mutual Exclusion

- A section or consecutive lines of code which should be seen as atomic is called critical section
- Synchronization which is needed for critical section is called mutual exclusion



Busy Waiting – Solution for Mutual Exclusion

- Busy waiting is one of the solution to implement synchronization (and a mutual exclusion)
  - Threads writes and reads shared variable flag
  - Thread intent to enter critical section must check flag. If it is false then it can't enter critical section and continuously checks flag to turn true in loop (busy waiting)
  - When it finds flags as turned to true and set flag to false and enters the critical section
  - Upon finishing the critical section code thread turns the flag to true again

Threads T1 and T2 execute the following code:

```
While flag == false
        do nothing
flag=false
enter critical section
…
finish critical section
flag = true
```

Problems with Busy Waiting

- Occupies CPU resources and waste precious CPU cycles
- Generates excessive traffic on bus or internal network
- Race Condition (Why?)
  - Think what would happen if both threads T1 and T2 intent to check the flag at exactly the same time and found that the flag is false. Both will be allowed to enter critical section which violate mutual exclusion rule

Suspend and Resume - – Solution for Busy Waiting

- Instead of waiting the loop continuously thread can be suspended when cant enter the critical section
- It can be resumed (woken up) when it can enter the critical section (Generally done through sending signal)
- For example, assume that T2 has already entered critical section hence the flag is false. When T1 tries to enter critical section and checks flag which is false it is suspended. After finishing critical section T2 will resume T1 by sending signal
- Disadvantage: Still leads to race condition Why ?
  - Flag is a shared variable which is used to control the action
  - Testing and action (suspend) should be done atomically

Threads T1 and T2 execute this code:

```
if flag == false
        suspend T?
flag=false
enter critical section
…
finish critical section
flag = true
resume T?
```

Mutual Exclusion – Solution – Not possible in multi-core environment Why?

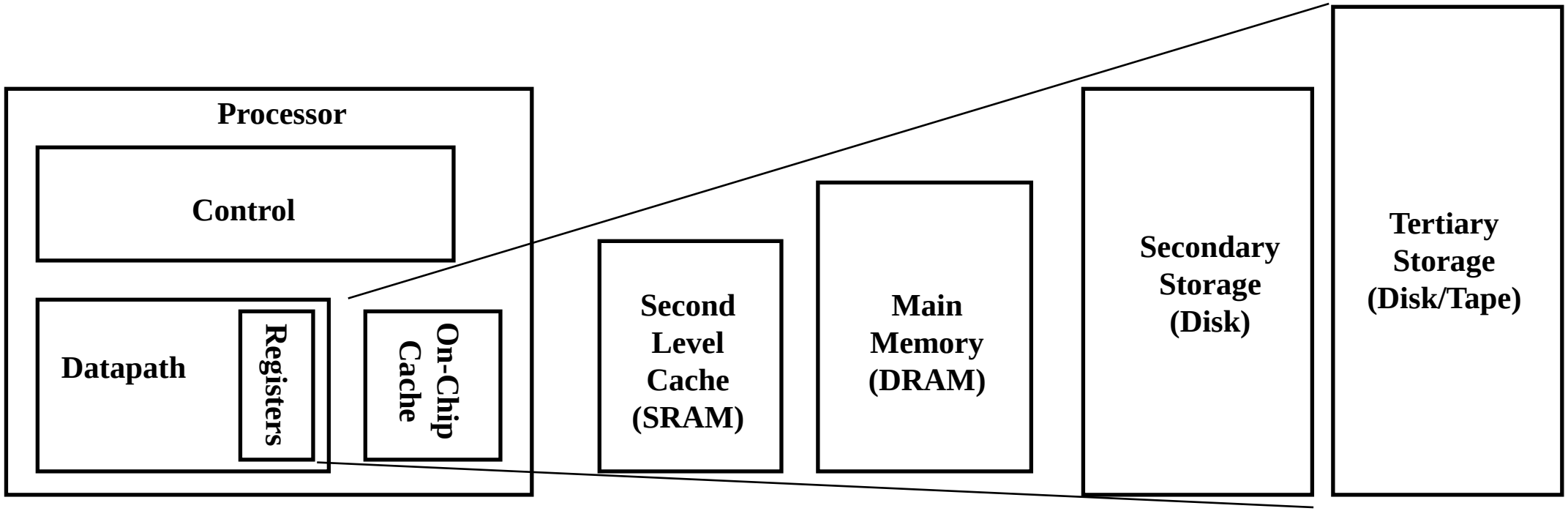
```
Thread T1
  loop
    flag1 := up
    turn := 2
    while flag2 = up and turn
= 2 do
      null
    end
    critical section
    flag1 := down
    non-critical section
  end
end T1
```

```
Thread T2
  loop
    flag2 := up
    turn := 1
    while flag1 = up and
turn = 1 do
      null
    end
    critical section
    flag2 := down
    non-critical section
  end
end T2
```

- Now we can guarantee that both the threads never enters the critical section for an infinite time
- And thread which starts its pre-protocol will eventually enter the critical section regardless of behavior of other thread

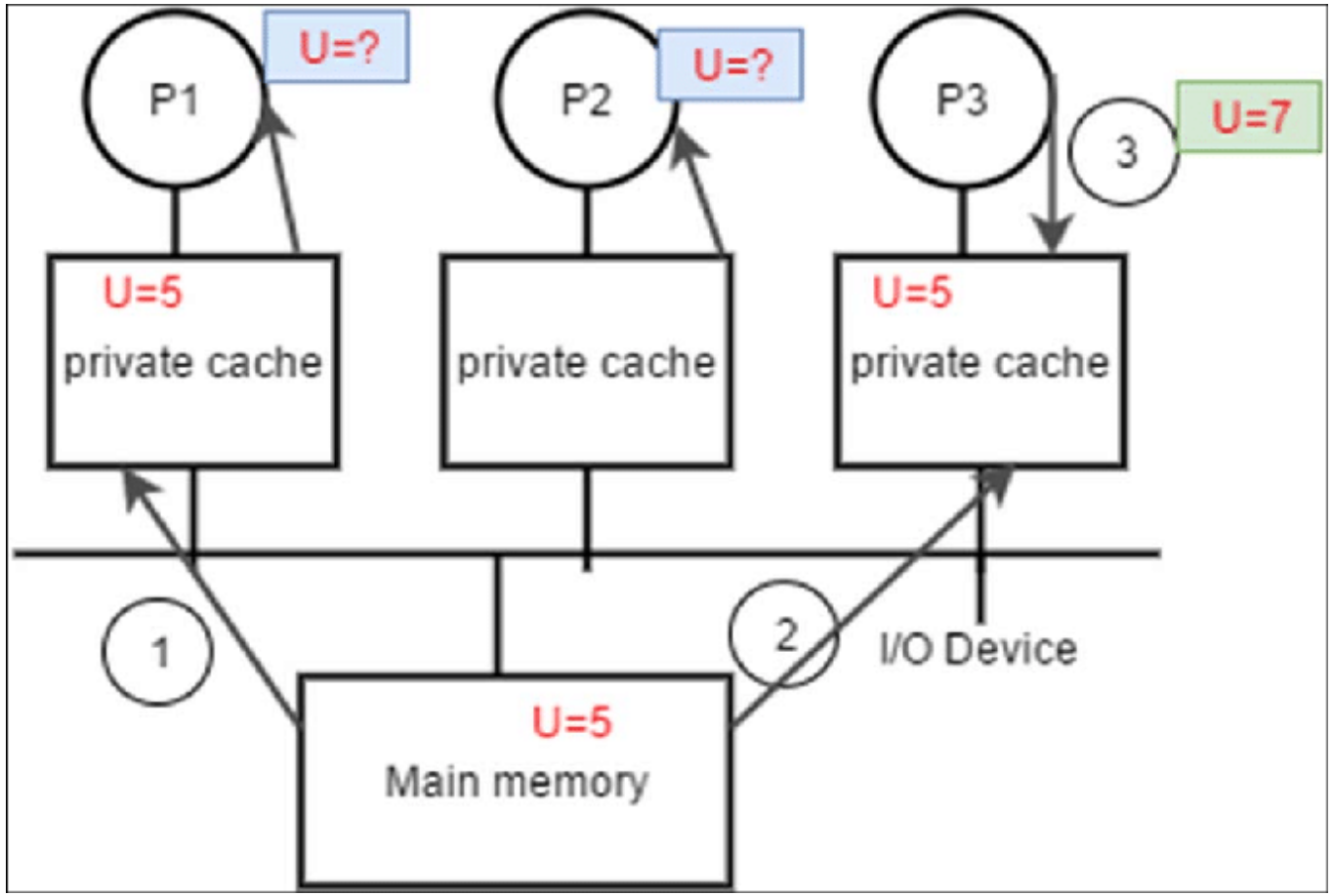
A Modern Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



Cache Coherence Problem

1. P1 reads value of U variable from main memory to private cache
2. P3 reads value of U variable from main memory to private cache
3. P3 updates value of U to 7 in private cache
4. Just before P3 gets a chance to update U in main memory, P2 reads from main memory and P1 from private cache has value of 5 but current value should have been 7



Cache Coherency Problem Solution

- Processor (P3) updating the value performs atomic update to cache and memory at the same time.
- Lock the system bus for all other processors (P1 and P2) so they cannot read or write until the data is written by processor (P3) who is updating the value.
- Make the private cache values as dirty for all other processors (P1 and P2) so that subsequent read must come from main memory only

MUTEX implementation in today's Multi-core System

```
static __always_inline bool __mutex_trylock_fast(struct mutex *lock) { unsigned long curr = (unsigned long)current; unsigned long zero = 0UL;

 if (atomic_long_try_cmpxchg_acquire(&lock->owner, &zero, curr)) return true; return false; }


#define atomic_long_try_cmpxchg_acquire(l, old, new) \
(ATOMIC_LONG_PFX(_try_cmpxchg_acquire)((ATOMIC_LONG_PFX(_t) *)(l), \ (ATOMIC_LONG_TYPE *)(old), (ATOMIC_LONG_TYPE)(new)))


#define ATOMIC_LONG_PFX(x) atomic64 ## x
```

ConcurrentProgramming\test_atomic.c

Make file to create object file

ConcurrentProgramming\test_atomic_makefile

Copy make file as Makefile and run make to create object files

Now create assembly file →

objdump -d -S test_atomic.ko > test_atomic.assembly

MUTEX implementation in today's Multi-core System

```
 10:    c7 00 02 00 00 00       movl   $0x2,(%rax)
16:     48 8b 05 00 00 00 00    mov    0x0(%rip),%rax       # 1d <init_module+0x1d>
1d:     f0 83 00 02             lock addl $0x2,(%rax)
21:     5d                      pop    %rbp
```

lock instruction tell processor running this code to lock the data bus so that other processor cannot perform any read or write operation. In other words lock instruction is implemented at hardware level

Mutual Exclusion Implementation in POSIX Thread - MUTEX

MUTEX is like a key to access the critical section that has access to only one thread at a time

#include <pthread.h>

MUTEX variable (containing union of structures) is represented as pthread_mutex_t data type defined in /usr/include/bits/pthreadtypes.h

Before we can use mutex variable memory allocation has to be done for which use function:

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) → return 0 on success

When mutex variable is no longer required memory should be freed for which we use function:

int pthread_mutex_destroy(pthread_mutex_t *mutex) → return 0 on success

Mutual Exclusion Implementation in POSIX Thread - MUTEX

#include <pthread.h>

To lock mutex: if mutex is already locked by another thread, thread trying to load mutex will be blocked

int pthread_mutex_lock(pthread_mutex_t *mutex) → return 0 on success

To lock mutex: if mutex is already locked by another thread, function will rerurn error code EBUSY

int pthread_mutex_trylock(pthread_mutex_t *mutex) → return 0 on success

To unlock mutex:

int pthread_mutex_unlock(pthread_mutex_t *mutex) → return 0 on success

Avoidance of Race Condition using MUTEX

ConcurrentProgramming\thread_racecond_without_mutex.c

Race condition output without using MUTEX (remove all pthread_mutex_* function calls)

$ ./thread_racecond.out → Race condition exists without using MUTEX
12345678901abcde23fgh456789012345ij6789012345678kl90mnopqrstuvwxyzabcdefghijkl
mnopqrstuvwxyz

ConcurrentProgramming\thread_racecond_with_mutex.c

With MUTEX implementation never get race condition

$ ./thread_racecond.out
12345678901234567890123456789012345678901234567890abcdefghijklmnopqrstuvwxyzabcdefghijkl
mnopqrstuvwxyz

OR

$ ./thread_racecond.out
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz12345678901234567890123456
78901234567890

Disadvantage of MUTEX

- Deadlock can occur if:
  - Thread locking the same MUTEX twice
  - Thread1 holding lock on mutex1 wants to get lock on mutex2 while thread2 holding lock on mutex2 wants to get lock on mutex1
  - Solution: As much as possible use pthread_mutex_trylock function
- Only one thread is allowed to lock mutex but in some situations such as Reader-Write problem or Dinning-Philosopher problem multiple locks should be possible
  - Solution: Reader-Writer Locks and Semaphores

Reader-Writer Locks

- Multiple readers should be allowed to have a lock on mutex
- Only one writer is allowed to have a lock on mutex
- Reading and Writing activities are mutually exclusive i.e. if reader lock is already there writer is not allowed and vice a versa
- In most implementation, if writer requesting lock is waiting due mutex which already has a reader lock then reader requests after the writer request will also wait so that writer is not waiting indefinitely (known as starvation problem)

#include <pthread.h>

To initialize use i.e. allocate memory:

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr)

To free the allocated memory: It will destroy the lock even if there are already locks on it so before destroy application should keep track of all the locks and unlock them before calling destroy

int pthread_rwlock_destroy(pthread_rwlock_t *restrict rwlock)

Reader-Writer Locks

#include <pthread.h>

For requesting read lock: reader will be blocked if there is already a write lock on mutex

int pthread_rwlock_rdlock(pthread_rwlock_t *restrict rwlock)

For requesting write lock: writer will be blocked if there is already at least one read lock or another write lock on mutex

int pthread_rwlock_wrlock(pthread_rwlock_t *restrict rwlock)

For requesting unlock: works for both read or write. If multiple reader locks are held, you have to initiate the same number of unlock for release (i.e. only the last unlock will release the lock otherwise it remains in reader lock state)

int pthread_rwlock_unlock(pthread_rwlock_t *restrict rwlock)

Non-blocking functions:

int pthread_rwlock_tryrdlock(pthread_rwlock_t *restrict rwlock)

int pthread_rwlock_trywrlock(pthread_rwlock_t *restrict rwlock)

ConcurrentProgramming\reader_writer.c

Reader-Writer Locks Usage

- Producer-Consumer Problem
- Several Operating System Data Structures:
  - Memory Page Tables
  - Process Control Blocks
  - Kernel Thread Tables
  - etc.

Dining-Philosopher Problem

- Five philosophers spend their time eating and thinking.
- They are sitting in front of a round table with spaghetti served.
- There are five plates at the table and five chopsticks set between the plates.
- Eating the spaghetti requires the use of two chopsticks which the philosophers pick up one at a time.
- Philosophers do not talk to each other.
- Semaphore chopstick [5] initialized to 1

Semaphores

- Allow multiple locks
- Semaphore S → integer variable
- Modified by two operations → wait() and signal()
- wait() – originally called P() for Dutch word “proberen” which means try

```
wait(S) {
        while S <= 0; // no-op
        S--;
}
```

- signal() – originally called V() for Dutch word “verhogen” which means increase

```
signal(S) {
        S++;
}
```

- Note wait() and signal() are atomic operation
- Semaphore Types:
- Binary Semaphore: S value can be 0 (locked) or 1 (unlocked)
- Counting Semaphores: S value can any integer
  - S=0 → locked by at least one thread
  - S=-n → n threads are waiting to acquire a lock
  - S=n → n locks are available for threads

Semaphore Usage

- Binary Semaphore in place of mutex
- Counting Semaphore used when multiple resources are to be shared with multiple consumers
  - Processes or threads sharing processors
  - Processes or threads sharing multiple network devices
  - Courses sharing classrooms
  - Faculties sharing office computers
  - etc.

Thread Synchronization using Semaphores Example of Binary Semaphore

```
//Thread1:
int t;
wait(sem)
sum = sum + x;
t = sum;
…
signal(sem);
```

```
//Thread2:
int t;
wait(sem)
sum = sum + y;
t = sum;
…
signal(sem);
```

Dining-Philosopher Problem: Solution

The structure of Philosopher i:

```
do {
  wait ( chopstick[i] ); //lock
  wait ( chopStick[ (i + 1) % 5] ); //lock
  // eat
  signal ( chopstick[i] ); //unlock
  signal (chopstick[ (i + 1) % 5] );
//unlock
  // think
} while (true) ;
```

- Problem is deadlock : when all philosopher decides to eat at the same time each will pick up one of the 2 chopsticks they need and wait for the other
- Solution: Allow to lock if both chopsticks are available at the same time
  - Lock the 1st chopstick
  - For 2nd chopstick, check if it can be locked otherwise release the 1st one

POSIX: Semaphores

- creating a semaphore:

int sem_init(sem_t *sem, int pshared, unsigned int value);

  - initializes a semaphore object pointed to by sem to integer value of “value”
  - pshared is a sharing option; a value of 0 means the semaphore is local to the calling process i.e. shared by all threads of the same process; a +ve number indicates it can be shared across multiple processes using shared memory instructions

- terminating a semaphore:

int sem_destroy(sem_t *sem);

  - frees the resources allocated to the semaphore sem
  - usually called after pthread_join()
  - an error will occur if a semaphore is destroyed for which a thread is waiting

POSIX: Semaphores

- semaphore control:

int sem_post(sem_t *sem); → same as signal()

sem_post atomically increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)

int sem_wait(sem_t *sem);

sem_wait atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

Semaphore Example

1 Resource, 1 semaphore with 1 thread

ConcurrentProgramming\semaphore1.c

$ ./semaphore1.out

Starting thread, semaphore is unlocked.

Hello from thread!

Hello from thread!

Semaphore locked.

Semaphore unlocked.

Hello from thread!

$

- ConcurrentProgramming\semaphore2.c
- 2 Semaphores and 2 Resources with 4 threads

Dining-Philosopher Problem: Solution

ConcurrentProgramming\dining-philosopher.c

Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 4 is Hungry
Philosopher 5 is Hungry
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
…….

Additional Slides

Mutual Exclusion

- Problem Statement:
- Suppose we have 2 threads with structure on the right side
- In which way the protocol can be implemented so that we can guarantee the mutual exclusion?

```
Thread T
  loop
    entry protocol
          critical section
    exit protocol
    non-critical section
  end
End T
```

Mutual Exclusion – Problem 1

```
Thread T1
  loop
    flag1 := up
    while flag2 = up do
      null
    end
    critical section
    flag1 := down
    non-critical section
  end
end T1
```

```
Thread T2
  loop
    flag2 := up
    while flag1 = up do
      null
    end
    critical section
    flag2 := down
    non-critical section
  end
end T2
```

What is the problem here?

- T1 sets flag1 := up and at the same time T2 sets flag2 := up
- T2 test flag1 and goes into busy-wait
- T1 test flag2 and goes into busy-wait
- Problem : livelock
- In this case both will go into busy-wait
- Why? None of the threads checks upfront if it is possible to enter the critical section

Mutual Exclusion – Solution

```
Thread T1
  loop
    while flag2 = up do
      null
    end
    flag1 := up
   critical section
    flag1 := down
    non-critical section
  end
end T1
```

```
Thread T2
  loop
    while flag1 = up do
      null
    end
    flag2 := up
    critical section
    flag2 := down
    non-critical section
  end
end T2
```

What is the problem now?

- T1 and T2 not in critical section (flag1 := down and flag2 := down)
- T2 test flag1 → found down
- T1 test flag2 → found down
- T2 sets flag2 := up and executes critical section
- T1 sets flag1 := up and executes critical section
- Problem : T1 and T2 both are in critical section (FAULT)
- Why? Testing of flag of other thread and setting its own flag are 2 individual action but they need to be atomic