Operating Systems 2023 Spring Term
Week 5

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

Threads II - Synchronization I

April 6, 2023

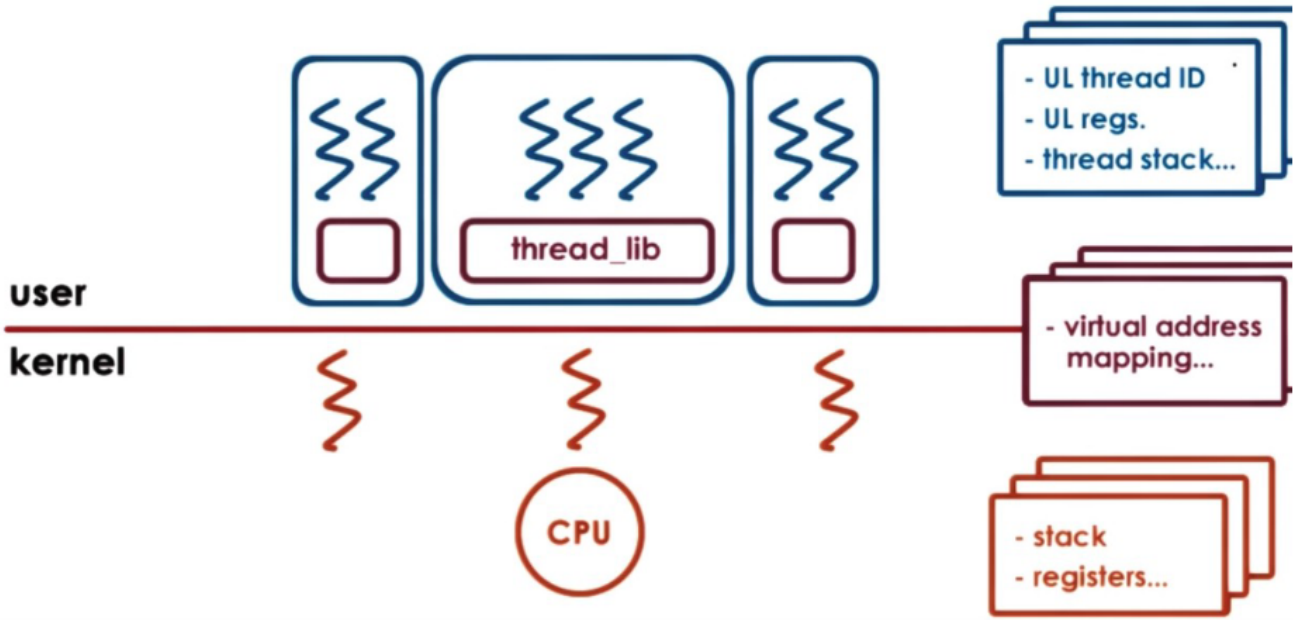
Week 4: Sample Glossary

- **race condition:** A situation in which two threads are concurrently trying to change the value of a variable. (on Page 1265)
- **mutual exclusion:** A property according to which only one thread or process can be executing code at once. (on Page 1259)
- **named pipes:** A connection-oriented messaging mechanism—e.g., allowing processes to communicate within a single computer system. (on Page 1259)
- **message passing:** In interprocess communication, a method of sharing data in which messages are sent and received by processes. Packets of information in predefined formats are moved between processes or between computers. (on Page 1257)

User Threads and Kernel Threads

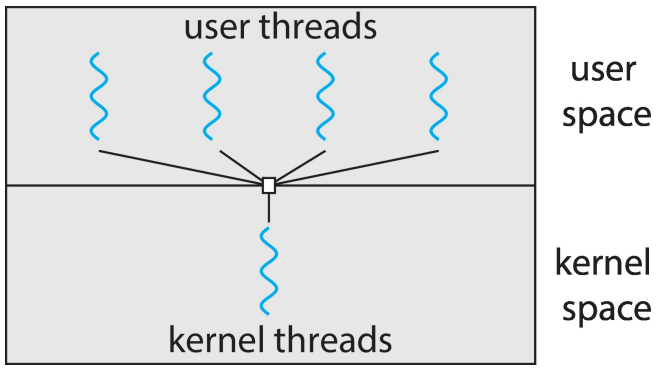
- User threads - management done by user-level threads library
- Three primary thread libraries:
  POSIX Pthreads
  Windows threads
  Java threads
- Kernel threads - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  Windows
  Linux
  Mac OS X
  iOS
  Android

User Threads and Kernel Threads



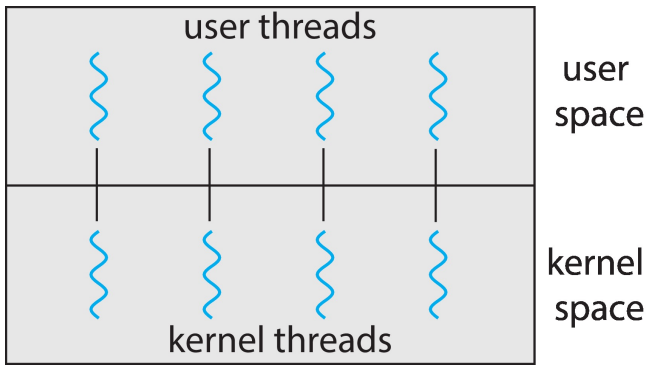
Multithreading Models: Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model - Examples: (1) Solaris Green Threads (2) GNU Portable Threads
  + Does not depend on OS
  - OS has no insights



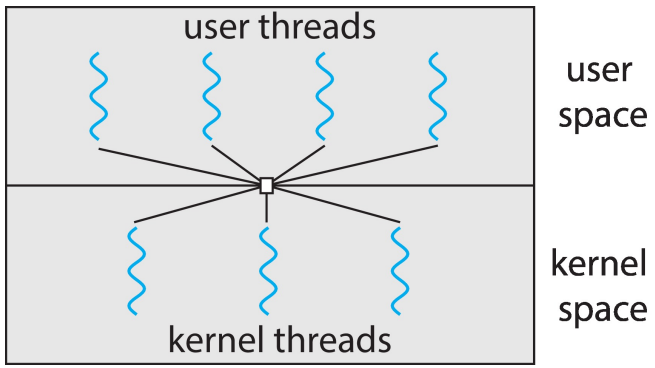
Multithreading Models: One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead - Examples (1) Windows (2) Linux
  + OS sees/understands threads
  - Must go to OS for all operations



Multithreading Models: Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the ThreadFiber package
- Otherwise not very common
  + Can be the best of both worlds
  - Requires coordination between user and kernel level thread managers



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - **Library entirely in user space:** All code and data structures exist in user space -> invoking a function in the library results in a local function call in user space and not a system call
  - **Kernel-level library supported by the OS:** code and data structures exist in kernel space -> invoking a function in the API results in a system call to the kernel

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Specification, not implementation
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux Mac OS X)
- Example: (A multithreaded program performing the summation of a non-negative integer in a separate thread)

$$sum = \sum_{i=1}^{N} i \tag{1}$$

Pthreads Example I

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}
```

Pthreads Example II

The program has two threads: (1) parent thread in main() and (2) summation or child thread performing the summation operation in the runner() function

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

After creating the summation thread, the parent thread will wait for it to terminate by invoking the pthread_join()
The summation thread will terminate when it calls pthread_exit()

Pthreads Example III

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Thread pools, compiler directives (OpenMP)

Thread Pools

- Create a number of threads in a pool where they await work
- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool
- Separating task to be performed from mechanics of creating task
- Python: "from multiprocessing import Pool" (https://docs.python.org/3/library/multiprocessing.html)

OpenMP

- OpenMP is an API for shared-memory parallel programming
  “directives-based” shared-memory API
- Identifies parallel regions – blocks of code that can run in parallel
  #pragma omp parallel
- Create as many threads as there are cores
  #pragma omp parallel for
  for(i=0;i<N;i++) {
  c[i] = a[i] + b[i];
  }
- Run for loop in parallel

Pthreads vs OpenMP

- Pthreads is lower level, requires that the programmer explicitly specify the behavior of each thread
- OpenMP sometimes allows the programmer to simply state that a block of code should be executed in parallel, and the precise determination of the tasks and which thread should execute them is left to the compiler and the runtime system

Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through clone() system call
  clone() allows a child task to share the address space of the parent task (process)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Process Synchronisation

- Background
  - race condition (concurrent access to shared data may result in data inconsistency)
- The Critical-Section Problem (to ensure the consistency of shared data)
- Peterson's Solution
- Synchronisation Hardware

Background

- Processes can execute concurrently (May be interrupted at any time, partially completing execution)
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer counter that keeps track of the number of full buffers. Initially, counter is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race Condition

`counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Critical Section (CS) Problem

- Consider system of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has critical section segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section

Critical Section: General structure of process P_i

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```

Algorithm for Process Pi

```
do {

    while (turn == j);

            critical section

    turn = j;

            remainder section

} while (true);
```

Solution to Critical-Section Problem

1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   Assume that each process executes at a nonzero speed
   No assumption concerning relative speed of the n processes

Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
  Preemptive – allows preemption of process when running in kernel mode
  Non-preemptive – runs until exits kernel mode, blocks, or voluntarily yields CPU
  Essentially free of race conditions in kernel mode

Peterson's Solution I

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  int turn;
  Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!

Peterson's Solution: Algorithm for Process Pi

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
} while (true);
```

Peterson's Solution II

Provable that the three Critical Section (CS) requirement are met:

1. Mutual exclusion is preserved
   Pi enters CS only if:
   either flag[j] = false or turn = i
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking
  Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  Currently running code would execute without preemption
  Generally too inefficient on multiprocessor systems
  Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  Atomic = non-interruptible
  Either test memory word and set value
  Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

test_and_set Instruction

```
boolean test_and_set (boolean *target)
  {
       boolean rv = *target;
       *target = TRUE;
       return rv;
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

```
do {
       while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
 } while (true);
```

- Shared Boolean variable lock, initialized to FALSE

compare_and_swap Instruction

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter value
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */
    /* critical section */
 lock = 0;
    /* remainder section */
} while (true);
```

- Shared integer “lock” initialized to 0;

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```