SWE2024

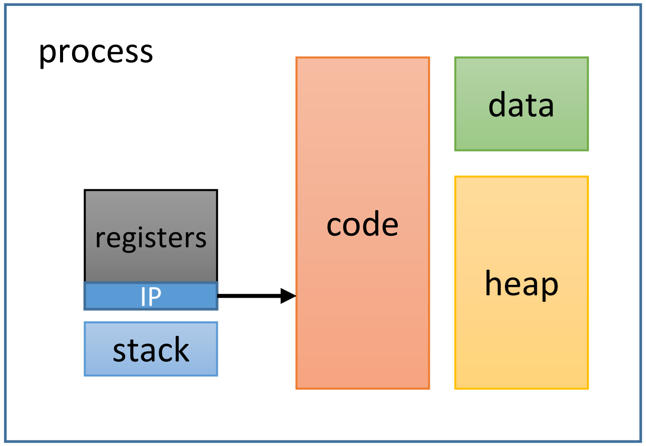
SUNGKYUNKWAN UNIVERSITY

Pthreads

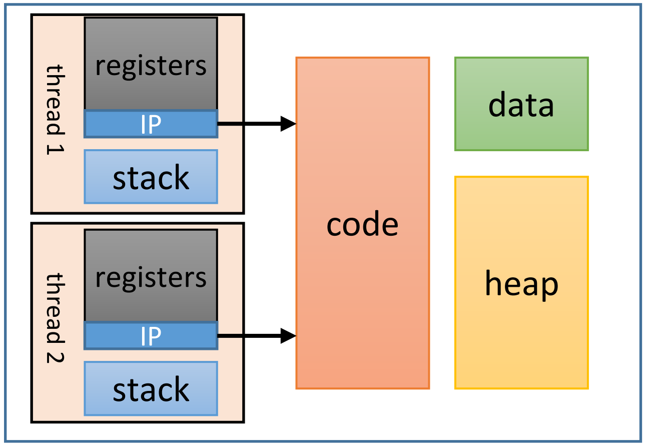
Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

Thread vs. Process

- Process
  - Process has its own address space
  - Each process has its own data
    - e.g., global variables, stack, heap



- Thread
  - Multiple thread share on address space
    - But its own stack and register context
  - Threads within same address space share data
    - e.g., global variables, heap



POSIX Threads

- Pthreads = POSIX thread

- Standard thread APIs for UNIX
  - IEEE standard 1003.1c-1995
  - Allow program to control multiple threads that overlap in time

Pthreads APIs

- Thread management APIs
  - Create, terminate, and join threads
  - Set and query thread attributes
- Mutex APIs
  - Create, destroy, lock, and unlock mutexes
- Conditional variable APIs
  - Create and destroy condition variables
  - Wait and signal based upon specified variable values

Pthreads Usage

- Pthreads header
  - Header file (*pthread.h*) is required to use Pthreads APIs

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

- Compilation
  - Almost all compilers support compilation with Pthreads library
  - In GNU compiler, add *–lpthread* to compile Pthreads programs

```
$ gcc –Wall –o hello hello.c –lpthread
```

Data Types of Pthreads

- **pthread[_object]_t**
  - **pthread_t**
    - Handle of a thread
    - Contain thread ID after creating thread
  - **pthread_attr_t**
    - Handle thread attribute type
    - Set attribute of newly created thread
  - **pthread_mutex_t**, **pthread_mutexattr_t**
  - **pthread_cond_t**, **pthread_condattr_t**
  - etc.

Thread Creation

- **int pthread_create (pthread_t *thread,**
  pthread_attr_t *attr,
  void *(*start_routine)(void*),
  void *arg)
  - Return new **thread ID** via *thread* parameter
    - Thread ID is useful for various operations on thread
  - *attr* parameter is used to set thread attributes
    - NULL for default values
  - *start_routine* parameter denotes the **C routine**
    - Thread will execute once it is created
  - A single argument may be passed to *start_routine* via *arg*
    - For multiple *args*, store them in a structure and pass its pointer
  - Return 0, on successful thread creation
    - Non-zero if unsuccessful

Thread Termination

- **void pthread_exit (void *retval)**
  - Terminate execution of calling thread
    - After a thread has completed its work typically
    - A thread is no longer required to exist
  - *retval* parameter is return value of thread
    - This value can be consulted from other threads with **pthread_join()**
  - It does not close files
    - Any files opened inside the thread will remain open after thread is terminated

Thread Creation & Termination Example

```
#define NUM_THREADS 4

void *thread(void *arg) {
    long id = (long)arg;
    printf("thread#%ld: Hello Thread!\n", id);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid[NUM_THREADS];
    long t;

    for (t = 0; t < NUM_THREADS; t++) {
        printf("main: creating thread#%ld\n", t);
        if (pthread_create(&tid[t], NULL, thread, (void*)t)) {
            printf("ERROR: pthread creation failed.\n");
            exit(1);
        }
    }
    printf("main: bye bye!\n");
    pthread_exit(NULL);
}
```

Thread Joining

- **int pthread_join (pthread_t thread, void **retval)**
  - Suspend execution of caller thread
    - Until the thread identified by *thread* parameter terminates
    - Thread terminates by
      - ✓ Calling **pthread_exit()**
      - ✓ Cancelled from other thread (such as **pthread_cancel()**)
  - Return value is stored in the location pointed by *retval* parameter
    - **PTHREAD_CANCELLED** is stored if thread was cancelled
  - Thread must be joinable
    - It is impossible to join a detached thread
  - Returns 0 on success
    - Non-zero if unsuccessful

Thread Join Example

```
#define NUM_THREADS 4

void *thread(void *arg) {
    long id = (long)arg;
    printf("thread#%ld: Hello Thread!\n", id);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid[NUM_THREADS];
    long t;

    for (t = 0; t < NUM_THREADS; t++) {
        printf("main: creating thread#%ld\n", t);
        if (pthread_create(&tid[t], NULL, thread, (void*)t)) {
            printf("ERROR: pthread creation failed.\n");
            exit(1);
        }
    }
    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(tid[t], NULL);
    }
    printf("main: bye bye!\n");
    return 0;
}
```

Thread Detachment

- **int pthread_detach (pthread_t thread)**
  - Put thread in detached state
    - This guarantees that memory resources consumed by *thread* parameter will be freed immediately when thread terminates
    - However, this prevents other threads from synchronizing on termination of thread by calling **pthread_join()**
  - Return 0 on success
  - Thread can be detached when it is created:

```
pthread_t tid;
pthread_attr_t attr;

pthread_attr_init (&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, start_routine, NULL);
pthread_attr_destroy (&attr);
```

Thread Cancellation

- **int pthread_cancel (pthread_t thread)**
  - Send a cancellation request to a thread
    - This thread has same ID to *thread* parameter
  - Status of target thread determines the cancellation
    - **PTHREAD_CANCEL_ENABLE**: allow the cancellation
    - **PTHREAD_CANCEL_DISABLE**: disallow the cancellation; cancellation request is blocked until the thread can be cancelled
  - Type of target thread determines the reaction
    - **PTHREAD_CANCEL_ASYNCHRONOUS**: thread can be cancelled at anytime
    - **PTHREAD_CANCEL_DEFERRED**: cancellation request is deferred until the cancellation point
  - Return 0 if the cancellation successful
    - Non-zero if unsuccessful

Thread Identifiers

- **pthread_t pthread_self (void)**
  - – Return unique, system assigned thread ID of calling thread

- **int pthread_equal (pthread_t t1, pthread_t t2)**
  - – Return non-zero value if parameter t1 & t2 refer to same thread
  - – C language equivalence operator (==) should not be used to compare two thread IDs against each other
    - This is because thread IDs are opaque objects
      - ✓ Pthreads library identifies a unique thread using this object

Thread Attributes

- **pthread_attr_t**

- **int pthread_attr_init (pthread_attr_t *attr)**
  - Initialize attribute objects to their default value

- **int pthread_attr_destroy (pthread_attr_t *attr)**
  - Delete attribute objects initialized by **pthread_attr_init()**
  - Must be re-initialized to use destroyed object again

Thread Attributes (Cont.)

- **int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)**
  - Set attribute of thread as either detached or joinable
  - Value in *detachstate* parameter is applied to detach state attribute
    - e.g., **PTHREAD_CREATE_DETACHED**, **PTHREAD_CREATE_JOINABLE**

- **int pthread_attr_getdetachstate (pthread_attr_t *attr, int *detachstate)**
  - Retrieve thread attribute to find it is detached or joinable
  - Thread attribute state is returned in *detachstate* parameter

Thread-Local Storage (TLS)

- TLS is a computer programming method
  - It uses static or global memory local to a thread

- Efficient and flexible usage of thread-local data

- TLS in C language
  - Cooperation between compiler/linker and runtime system
  - In GNU C, use the new keyword __*thread*
    - Global variable – __**thread** int number;
    - Static variable – static __**thread** int number;

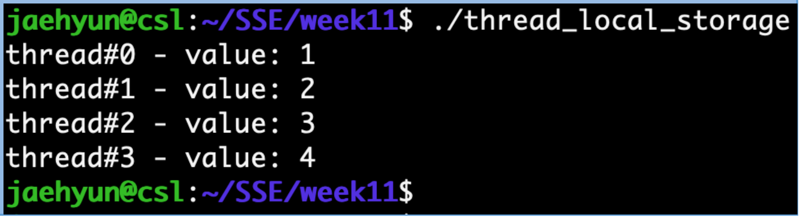
Thread-Local Storage Example

```c
#define NUM_THREADS 16

int value;

void *thread(void *arg) {
    long id = (long)arg;
    value++;
    printf("thread#%ld - value: %d\n", id, value);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid[NUM_THREADS];
    long t;
    for (t = 0; t < NUM_THREADS; t++) {
        if (pthread_create(&tid[t], NULL, thread, (void*)t)) {
            printf("ERROR: pthread creation failed.\n");
            exit(1);
        }
    }
    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(tid[t], NULL);
    }
    return 0;
}
```



```
jaehyun@csl:~/SSE/week11$ ./thread_local_storage
thread#0 - value: 1
thread#1 - value: 2
thread#2 - value: 3
thread#3 - value: 4
jaehyun@csl:~/SSE/week11$
```

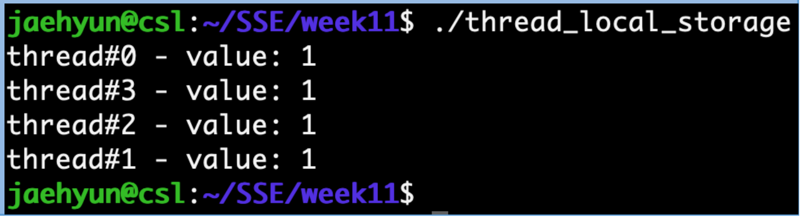
Thread-Local Storage Example

```
#define NUM_THREADS 16

__thread int value;

void *thread(void *arg) {
    long id = (long)arg;
    value++;
    printf("thread#%ld - value: %d\n", id, value);
    pthread_exit(NULL);
}

int main() {
    pthread_t tid[NUM_THREADS];
    long t;
    for (t = 0; t < NUM_THREADS; t++) {
        if (pthread_create(&tid[t], NULL, thread, (void*)t)) {
            printf("ERROR: pthread creation failed.\n");
            exit(1);
        }
    }
    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(tid[t], NULL);
    }
    return 0;
}
```



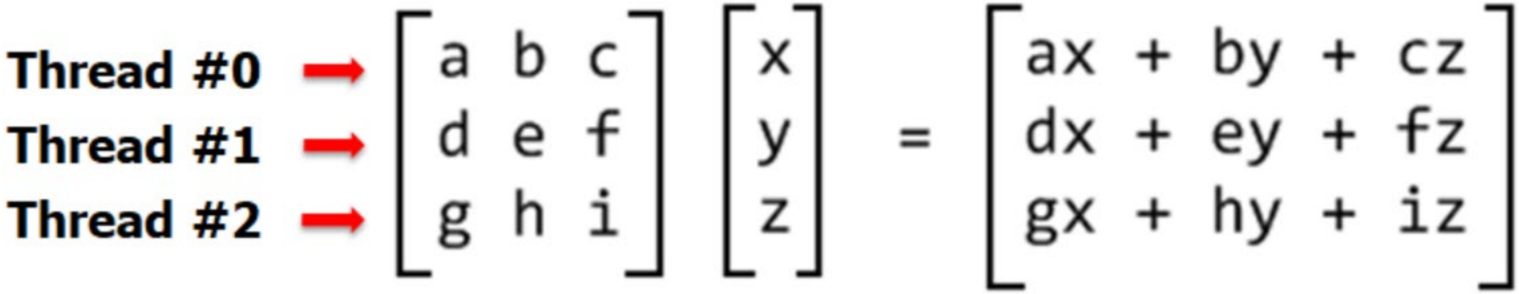
```
jaehyun@csl:~/SSE/week11$ ./thread_local_storage
thread#0 - value: 1
thread#3 - value: 1
thread#2 - value: 1
thread#1 - value: 1
jaehyun@csl:~/SSE/week11$ _
```

Exercise

- Matrix-vector multiplication using multi-threading
  - Multiplication result of **M*N matrix** and **N*1 vector** is **M*1 vector**
  - Row size (M), column size (N) of matrix are received by arguments
  - Elements of matrix and vector are randomly assigned between **0~9** (data type is **int**)
  - Create threads as much as the row size (M)
  - Each thread performs a calculation on one row of the matrix

Exercise

- Matrix-vector multiplication



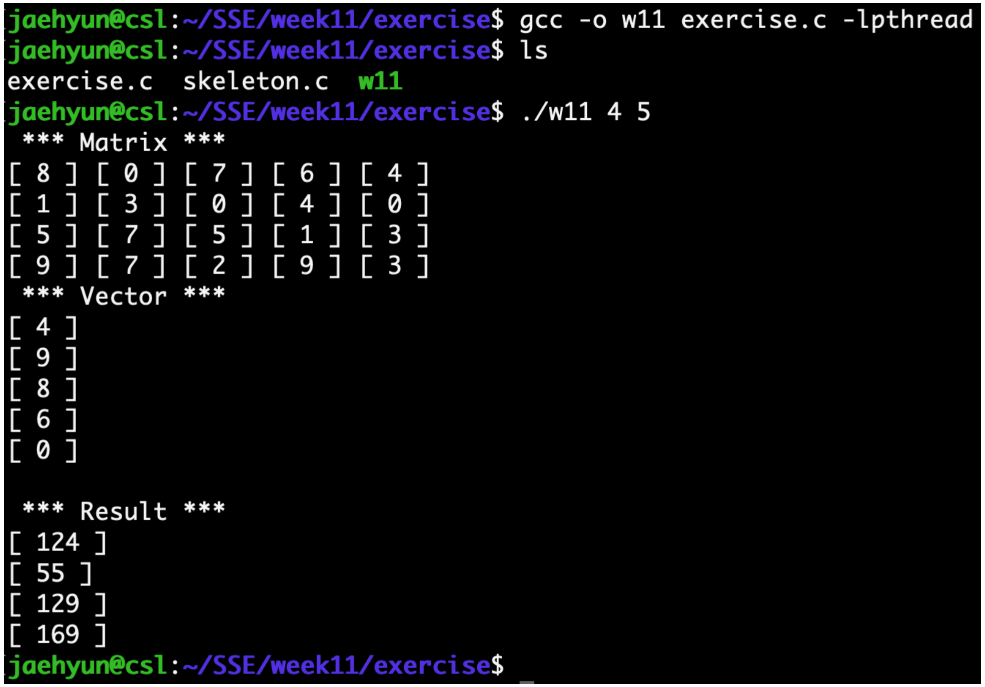
$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

- Restrictions
  - Thread ID and calculation results are delivered through the *thread_data* struct
    - Do not use global variable for thread ID and calculation result
  - Main function must wait other threads to terminate
    - Use **pthread_join()** system call

Exercise

- Random variable
  - Should include *time.h* header
  - srand(time(NULL));
  - int a = rand() % 10;

- Running example



```
jaehyun@csl:~/SSE/week11/exercise$ gcc -o w11 exercise.c -lpthread
jaehyun@csl:~/SSE/week11/exercise$ ls
exercise.c  skeleton.c  w11
jaehyun@csl:~/SSE/week11/exercise$ ./w11 4 5
 *** Matrix ***
[ 8 ] [ 0 ] [ 7 ] [ 6 ] [ 4 ]
[ 1 ] [ 3 ] [ 0 ] [ 4 ] [ 0 ]
[ 5 ] [ 7 ] [ 5 ] [ 1 ] [ 3 ]
[ 9 ] [ 7 ] [ 2 ] [ 9 ] [ 3 ]
 *** Vector ***
[ 4 ]
[ 9 ]
[ 8 ]
[ 6 ]
[ 0 ]

 *** Result ***
[ 124 ]
[ 55 ]
[ 129 ]
[ 169 ]
jaehyun@csl:~/SSE/week11/exercise$ _
```

Exercise

- Submit your exercise source code
  - To InUiYeJi Cluster
  - Put your Makefile and *.c files in p11 folder
  - Submit using

  $ ~swe2024-41_23s/bin/submit p11 p11
  - We will compile by using command *make*
    - When compilation fails, you get zero points
    - Compiled binary name should be “*p11*“

- Due 2023/5/12 23:59