# Threads II:
# POSIX API 'Pthreads'

Copyright ©: University of Illinois CS 241 Staff

# Passing Arguments to Threads

- **`pthread_create()`**
  - Only one argument to the thread start routine. It must be passed by reference and cast to **`(void *)`**
  - For multiple arguments
    - Create a structure that contains all of the arguments
    - Pass a pointer to that structure in **`pthread_create()`**

Copyright ©: University of Illinois CS 241 Staff

# Passing Arguments to Threads

Where should these be declared?

- Passing an int:
  - ```
    int i = 42;
    pthread_create(..., my_func, (void *)&i);
    ```
- Passing a C-string:
  - ```
    char *str = "UIUC";
    pthread_create(..., my_func, (void *)str);
    ```
- Passing an array:
  - ```
    int arr[100];
    pthread_create(..., my_func, (void *)arr);
    ```

Copyright ©: University of Illinois CS 241 Staff


    ```

# Passing Arguments to Threads

- Retrieving an int:
  - ```
    void *myfunc(void *vptr_value) {
        int value = *((int *)vptr_value);
    ```
- Retrieving a C-string:
  - ```
    void *myfunc(void *vptr_value) {
        char *str = (char *)vptr_value;
    ```
- Retrieving an array:
  - ```
    void *myfunc(void *vptr_value) {
        int *arr = (int *)vptr_value;
    ```

Copyright ©: University of Illinois CS 241 Staff


    ```

# Race Conditions

- What is a race condition?
  - "A race occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y."**
- Why do race conditions occur?
  - A race condition occurs when two or more threads can access/ modify shared data and the applied operations are non-atomic.
  - The scheduling algorithm can swap between threads at any point, interrupting a non-atomic operation and leaving the shared data in an inconsistent state. Hence, the result of the change in data is dependent on the thread scheduling algorithm.

**Definition from "Computer Systems: A Programmer's Perspective"



# Race Conditions

- Race conditions are notoriously difficult to debug, since they are often marginal (only occur in pathological and "very unlikely" situations), and highly dependent on the **relative timing** between interfering threads

- What solutions can we apply?
  - Use synchronization** primitives (like mutex/semaphore) when performing operations on shared data variables
  - Use non-preemptive scheduling (bad idea! ➔ **non-portable**)
  - Use atomic operations to modify global shared data (e.g., x86 instruction LOCK INC mem_location) (bad idea! ➔ **requires use of assembly code**)

*** We will study synchronization after scheduling!*

Copyright ©: University of Illinois CS 241 Staff


# Calling a library function: is it safe?

- Concurrent programming with pthreads can trigger race conditions or undefined behavior in your code if calling thread-unsafe functions
- POSIX pthreads: use library **thread-safe** functions
  - A thread-safe function is one that can be safely (i.e., it will deliver the same results regardless of whether it is) called from multiple threads at the same time. ➔ see man 7 pthreads
  - POSIX requires that all functions specified in the standard shall be thread-safe, except for the following functions (see next slide):

Copyright ©: University of Illinois CS 241 Staff


# System & library functions that are not required to be thread-safe

| | | | | | |
|---|---|---|---|---|---|
| asctime | dirname | getenv | getpwent | lgamma | readdir |
| basename | dlerror | getgrent | getpwnam | lgammaf | setenv |
| catgets | drand48 | getgrgid | getpwuid | lgammal | setgrent |
| crypt | ecvt | getgrnam | getservbyname | localeconv | setkey |
| ctime | encrypt | gethostbyaddr | getservbyport | localtime | setpwent |
| dbm_clearerr | endgrent | gethostbyname | getservent | lrand48 | setutxent |
| dbm_close | endpwent | gethostent | getutxent | mrand48 | strerror |
| dbm_delete | endutxent | getlogin | getutxid | nftw | strtok |
| dbm_error | fcvt | getnetbyaddr | getutxline | nl_langinfo | ttyname |
| dbm_fetch | ftw | getnetbyname | gmtime | ptsname | unsetenv |
| dbm_firstkey | gcvt | getnetent | hcreate | putc_unlocked | wcstombs |
| dbm_nextkey | getc_unlocked | getopt | hdestroy | putchar_unlocked | wctomb |
| dbm_open | getchar_unlocked | getprotobynumber | inet_ntoa | pututxline | |
| dbm_store | getdate | getprotoent | l64a | rand | |

Copyright ©: University of Illinois CS 241 Staff

# Classes of thread-unsafe functions

1. Functions that do not protect shared variables.

➔ thread-safety can be enforced by protecting the shared variables with synchronization operations

2. Functions that keep state across multiple invocations.

(see rand) ➔ thread-safety can be enforced by re-writing the function as reentrant; hence, the caller passes state information in the arguments

3. Functions that return a pointer to a static variable.

(see ctime) ➔ thread-safety can be enforced by re-writing the function as reentrant or using the "lock and copy" technique that associates a mutex to the unsafe function

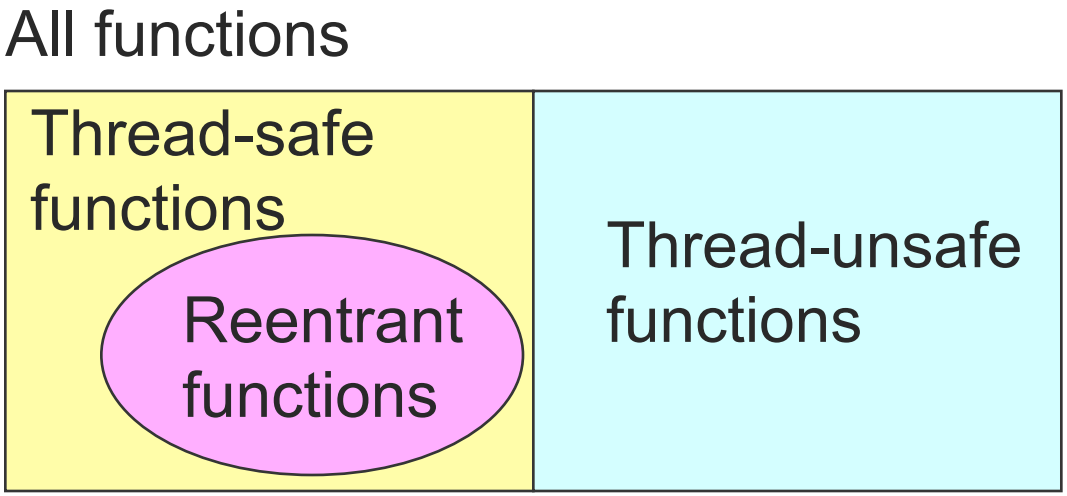
4. Functions that call thread-unsafe functions.

➔ use one of above solutions depending on class of called unsafe function

From "Computer Systems: A Programmer's Perspective"

# The set of all functions

- Relationship between the sets of reentrant, thread-safe, and non-thread-safe functions




From "Computer Systems: A Programmer's Perspective"

Copyright ©: Nahrstedt, Angrave, Abdelzaher

# Thread-safe

- Thread-safe means that a (library) function can be called from multiple threads without destructive results
- A way to make a function thread-safe is to protect its critical sections (accessing static or global non-constant data) with synchronization** primitives
- A more efficient solution is to re-write the library function as **reentrant** to become thread-safe
  - Every reentrant function is thread-safe; however, not every thread-safe function is reentrant.

** *We will study synchronization after scheduling!*

Copyright ©: Nahrstedt, Angrave, Abdelzaher

# Thread-safe

# Reentrant functions

- A function is described as reentrant if it can be reentered while it is already running (i.e. it can be safely executed concurrently)

- It is important to realize that reentrancy is a property that needs to be guaranteed by both the caller and callee.

➔ Improper use of a reentrant function will invalid this property (e.g., caller passes a shared variable as state information in the arguments)

- A reentrant function must:
  - hold no static (or global) non-constant data.
  - work only on the data provided to it by the caller.
  - not return a pointer to a static variable
  - not call non-reentrant routines.

Copyright ©: Nahrstedt, Angrave, Abdelzaher

# The case of rand()

- An example: rand and rand_r function
- int rand(void);
- int rand_r(unsigned int *nextp); ← reentrant version of rand()
- The function rand() is not reentrant or thread-safe, since it uses hidden state that is modified on each call. This might just be the seed value to be used by the next call, or it might be something more elaborate.
- In order to get reproducible behavior in a threaded application, this state must be made explicit. The function rand_r() is supplied with a pointer to an unsigned int, to be used to store state between calls.

# The case of rand()

```
unsigned int next = 1;

/* rand - return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

```
/* rand_r - a reentrant pseudo-random integer on 0..32767 */
int rand_r(unsigned int *nextp) {
    *nextp = *nextp * 1103515245 + 12345;
    return (unsigned int)(*nextp / 65536) % 32768;
}
```

From "Computer Systems: A Programmer's Perspective"



# Reverse_vector example

```
#define MAX_SIZE

static int tmpvector[MAX_SIZE];

int reverse_vector(int *vector, unsigned int size) {
  unsigned int idx;
  if(size > MAX_SIZE) return -1;
  for(idx = 0; idx < size; idx++)
    tmpvector[idx] = vector[size-idx-1];
  for(idx = 0; idx < size; idx++)
    vector[idx] = tmpvector[idx];
  return 0;
}
```

Is it thread-safe?

Copyright ©: University of Illinois CS 241 Staff

# Reverse_vector_r example

```
int reverse_vector_r(int *vector, unsigned int size) {

  int *startPtr, *endPtr;
  int tmp;
  startPtr = vector;
  endPtr = vector + size -1;
  while(startPtr < endPtr) {
    tmp = *endPtr;
    *endPtr = *startPtr;
    *startPtr = tmp;
    startPtr++; endPtr--;
  }
  return 0;
}
```

Is it thread-safe, reentrant or both?

Copyright ©: University of Illinois CS 241 Staff

# Useful when studying signals: what is async-signal-safe?

- POSIX signals: only call library **async-signal-safe** functions inside a signal handler. ➔ see man 7 signal

- A function is said to be **async-signal-safe** if it is either reentrant or non-interruptible by signals
  - You will learn more about it when discussing POSIX signals

Copyright ©: University of Illinois CS 241 Staff