Network Programming

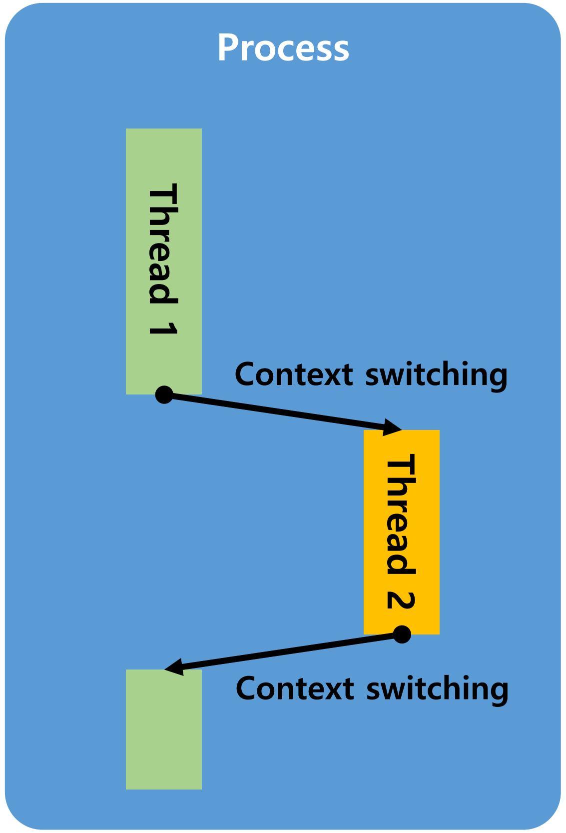
Thread Based Concurrent Programming

01

Thread

Thread

- The smallest sequence of programmed instructions
  - A part of a process
  - At least one thread is exists in a process (main thread)
  - Threads in a process can share resources of their process
- Managed independently by a scheduler
  - Each thread has its control flow and address space
  - Context switching is performed between threads
  - Threads can run concurrently



Traditional View of a Process

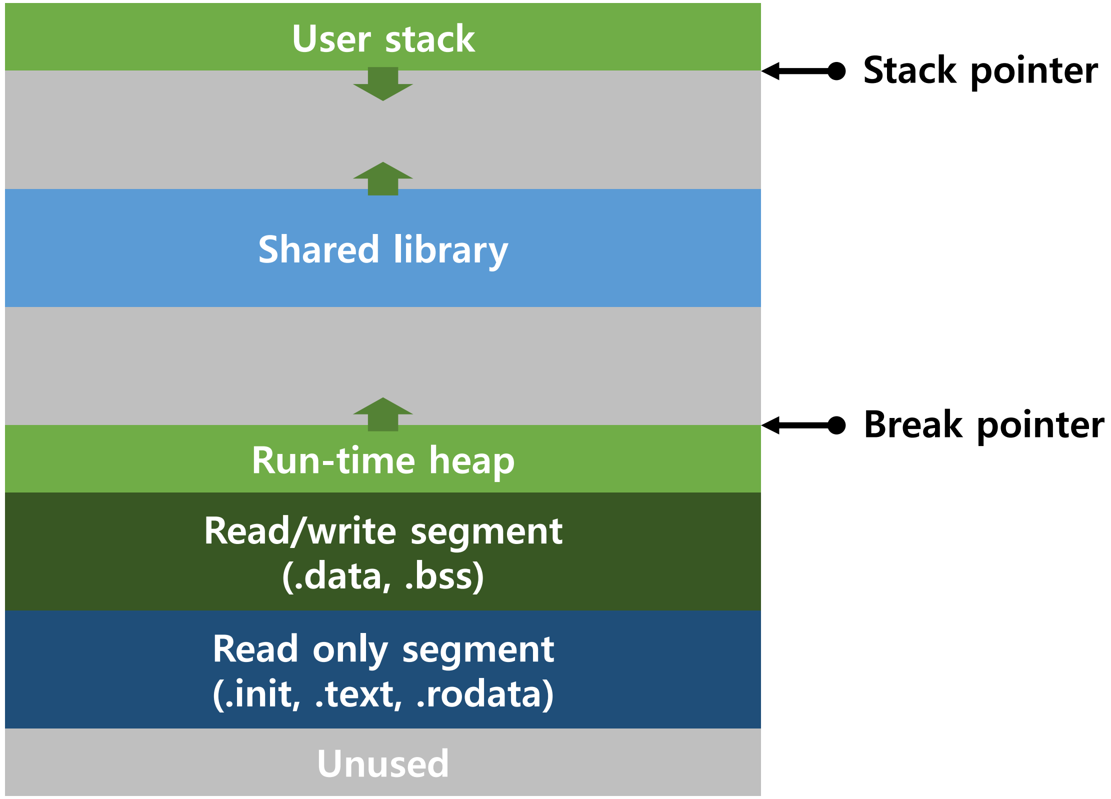
- Consist of process context, code, data, stack, and etc.

Process context

- Program context
    - Data register
    - Condition codes
    - Stack pointer
    - Program counter
- Kernel context
    - Virtual memory structure
    - Descriptor table
    - Break pointer

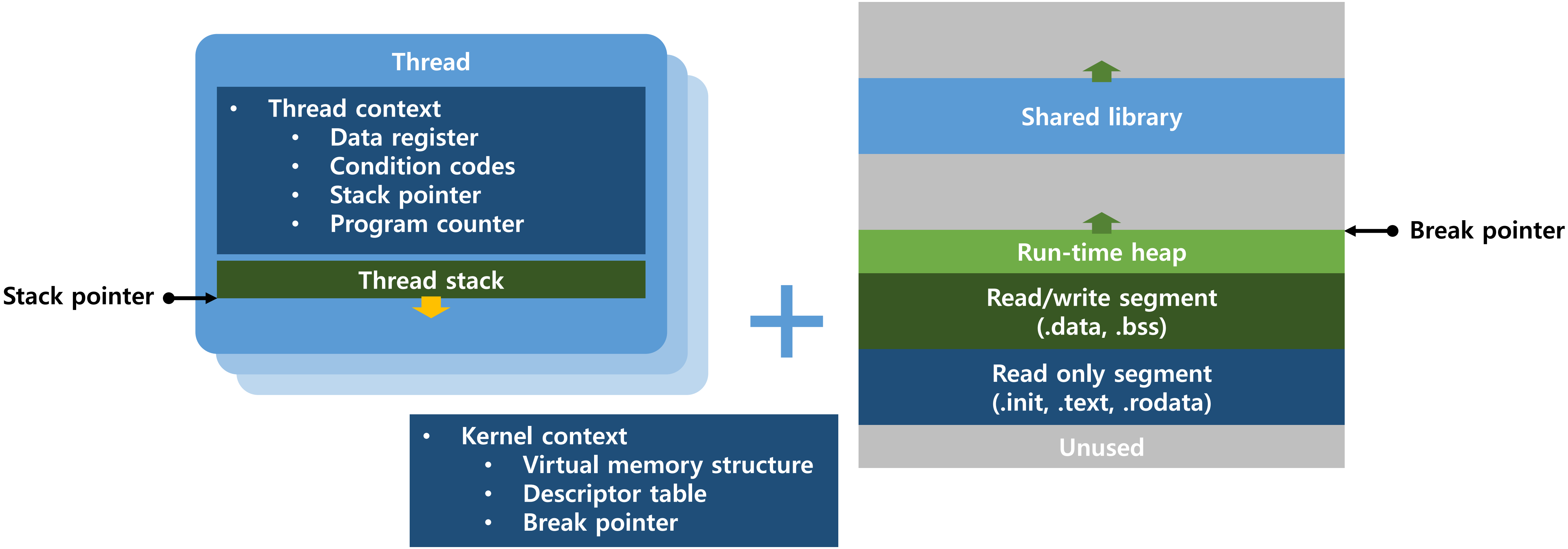
- Kernel context
    - Virtual memory structure
    - Descriptor table
    - Break pointer

+



Alternative View of a Process

- Consist of threads, kernel context, code, data, and etc.



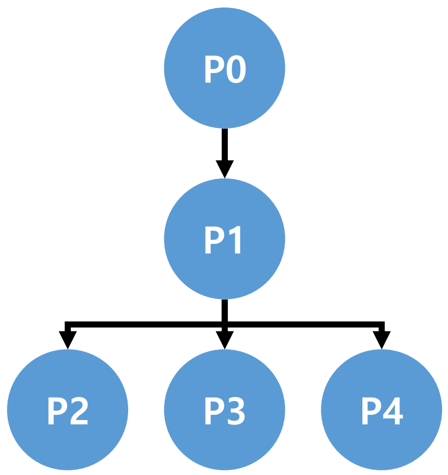
Alternative View of a Process

- Each thread has its own
  - Thread ID
  - Control flow
  - Stack and stack pointer

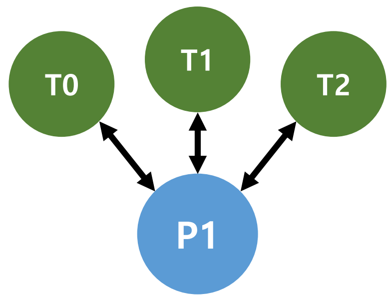
- Threads share
  - Kernel context of the process
  - Virtual memory address of the process

Logical View of Threads

- Processes form a tree hierarchy
- Threads associated with a process, form a pool of peers



Processes



Threads

Process and Thread

- Processes and threads are similar in
  - Each has its own ID
  - Each has its own control flow
  - Each has its own stack and stack pointer
  - Each can run concurrently
  - Each is context switched
- Processes and threads are different in
  - Threads can share resource of the process easily,
    - Processes can share resources too via IPC (inter process communication), but it’s hard
  - A thread are much more lighter than a process
    - Faster to create and reap

Posix Thread

- Posix
  - Portable operating system interface
  - Family of standards specified by the IEEE computer society
  - Maintaining compatibility between operating systems based on UNIX
  - Defines the API for operating systems based on UNIX

- Pthread
  - Standard thread interface for UNIX systems
  - About 60 functions are supported for C language

Posix Thread

- Posix
  - Portable operating system interface
  - Family of standards specified by the IEEE computer society
  - Maintaining compatibility between operating systems based on UNIX
  - Defines the API for operating systems based on UNIX

- Pthread
  - Standard thread interface for UNIX systems
  - About 60 functions are supported for C language

Posix Thread Creation

- Posix thread create function
  - int pthread_create(pthread_t* threadID, const pthread_attr_t* attribute,
    void* startRoutine, void* argumentPointer);
    - Create a new thread and run a routine with it
    - threadID: the ID of the new thread will be stored in here
    - attribute: attribute structure for the thread, usually NULL
    - startRoutine: a routine to run with the thread
    - argumentPointer: a pointer of arguments for the thread
    - Return 0 on success, non-zero on error

Posix Thread Joining

- Posix thread join function
  - int pthread_join(pthread_t* threadID, void** returnValuePointer);
    - Block the caller, and return when specified thread terminates

    - threadID: the ID of a thread to wait for
    - returnValuePointer: a return value of the thread will be stored in here

    - Return 0 on success, non-zero on error

Posix Thread Creation and Joining

```
…
# include <pthread.h>
// Add an option –lpthread to compile

void* threadRoutine(void* argumentPointer){
	pthread_t id = pthread_self();
	// Return the ID of the calling thread
	printf("Hello, this is thread %d.₩n", id);
	return NULL;
	// Thread should return something for join function
}

int main(){
	pthread_t threadID;

	pthread_create(&threadID, NULL, threadRoutine, NULL);
	pthread_join(threadID, NULL);

	return 0;
}
```

Passing Arguments to a Thread

```
...
void* threadRoutine(void* argumentPointer){
	argument = (argumentType) *argumentPointer;
	// Cast the argumentPointer into a proper type
	...
	// You can use the argument in here
}

int main(){
	pthread_t threadID;

	argumentType argument;
	// An argument of a type

	pthread_create(&threadID, NULL, threadRoutine, (void*) &argument);
	// Should pass the argument as a void pointer type
	// Of course you can just cast an integer into a void pointer type to pass
	...
}
```

Passing a Return Value to Join Function

```
...
void* threadRoutine(void* argumentPointer){
    ...
    int returnValue = 2016;
    return (void*)returnValue;
    // Should return a value as void pointer type
}

int main(){
    ...
    void* returnValuePointer;
    // Should be void pointer type
    pthread_join(threadID, &returnValuePointer);

    int returnValue = (int)returnValuePointer;
    // Cast the returnValuePointer into a proper type
    // Of course you can allocate a new return value than deallocate it
    ...
}
```

Posix Thread Detach

- Posix thread detach function
  - int pthread_detach(pthread_t threadID);
    - Detach the thread
    - Don't wait the thread with join function detached threads
    - When detached thread terminates, system reaps it automatically
    - threadID: the ID of a thread to detach
    - Return 0 on success, non-zero on error

Posix Thread Detach

```
…
# include <pthread.h>

…

int main(){
	pthread_t threadID;

	pthread_create(&threadID, NULL, threadRoutine, NULL);
	pthread_detach(threadID, NULL)

	…
	// Don't call a join function for the threadID

	return 0;
}
```

02

Memory Sharing

Memory Sharing

- Threads can share the virtual memory of their process
- Threads can not share each own stack
  - Note that the main function is a thread too
  - Hence, threads can share global variables of their process
- Threads can share static variables in each own stack

Memory Sharing Example

```
int globalVariable = 2016;

void* threadRoutine(void* argumentPointer){
	static int staticVariable = 1;
	int localVariable = pthread_self();
	…
}

int main(){
	…

	pthread_create(threadIDs[0], NULL, threadRoutine, NULL);
	pthread_create(threadIDs[1], NULL, threadRoutine, NULL);
	…
	int localVariable = 1027;
	int* localPointer = (int*)malloc(…);
	…
}
```

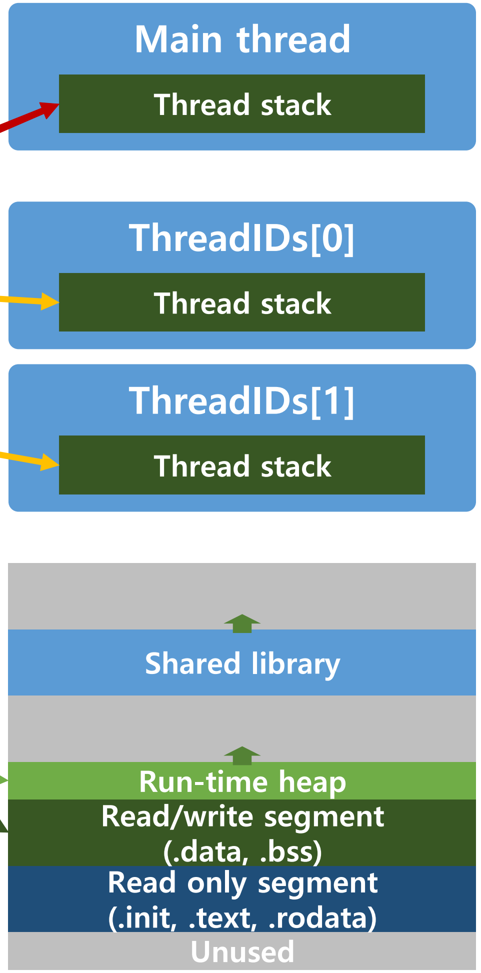
Memory Sharing Example

```
int globalVariable = 2016;

void* threadRoutine(void* argumentPointer){
    static int staticVariable = 1;
    int localVariable = pthread_self();
    ...
}

int main(){
    ...

    pthread_create(threadIDs[0], NULL, threadRoutine, NULL);
    pthread_create(threadIDs[1], NULL, threadRoutine, NULL);
    ...
    int localVariable = 1027;
    int* localPointer = (int*)malloc(...);
    ...
}
```



Memory Sharing Example

- Accessible table

	Main thread	threadIDs[0]	threadIDs[1]
globalVariable	O	O	O
staticVariable	X	O	O
localVariable (in threadIDs[0])	X	O	X
localVariable (in threadIDs[1])	X	X	O
localVariable (in main thread)	O	X	X
localPointer	O	X	X

Unsafe Region

- Memory sharing makes the application efficient
  - Much less redundant than using fork()
  - Faster to create and reap then processes

- However, it can be very dangerous
  - Memory sharing can make unsafe region in the application

Unsafe Region Example

```
int sum = 0;

void* threadRoutine(void* argumentPointer){
    for(int i = 0; i < 1000000; i++)
        sum++;
    …
}

int main(){
    …

    pthread_create(threadIDs[0], NULL, threadRoutine, NULL);
    pthread_create(threadIDs[1], NULL, threadRoutine, NULL);

    pthread_join(threadIDs[0], NULL);
    pthread_join(threadIDs[1], NULL);

    printf("sum: %d₩n", sum);
    …
}
```

Unsafe Region Example

- The result of sum should be 2,000,000
  - But it’s not (in a high probability)
- One line of C source code is not one instruction for CPU
  - In pseudo assembly code, sum++ is
    - load sum eax
    - increment eax
    - store sum eax

Unsafe Region Example

- If the threads run like below, the result must be 2,000,000

ThreadIDs[0]	ThreadIDs[1]
• load sum eax # sum = 10 • increment eax # sum = 10 • store sum eax # sum = 11	
	• load sum eax # sum = 11 • increment eax # sum = 11 • store sum eax # sum = 12

Unsafe Region Example

- However, threads don't run like above in a high probability
  - For example like below

ThreadIDs[0]	ThreadIDs[1]
• load sum eax # sum = 10 • increment eax # sum = 10	
	• load sum eax # sum = 10 • increment eax # sum = 10
• store sum eax # sum = 11	
	• store sum eax # sum = 11

Unsafe Region Example

```
int sum = 0;

void* threadRoutine(void* argumentPointer){
	for(int i = 0; i < 1000000; i++)
		sum++; // This area is unsafe region
	…
}

int main(){
	…

	pthread_create(threadIDs[0], NULL, threadRoutine, NULL);
	pthread_create(threadIDs[1], NULL, threadRoutine, NULL);

	pthread_join(threadIDs[0], NULL);
	pthread_join(threadIDs[1], NULL);

	printf(“sum: %d₩n”, sum);
	…
}
```

Synchronizing

- Synchronizing enforce concurrent flow to be sequential
  - Synchronizing unsafe region prevent unintended context switching
- Synchronizing consist of two steps
  - Wait
    - Check some conditions before run a synchronized region
    - If the conditions are not satisfied, wait for a wake up signal
    - When the signal is received, it wakes up and test some conditions again
  - Signaling
    - Send a wake up signal when exiting the synchronized region
- Two ways of multi-thread synchronizing
  - Mutual exclusion
  - Semaphore

Mutex

- Mutual exclusion
  - Set critical section to synchronize an unsafe region
  - Only one thread can run critical section
- Declaring a mutex
  - Use PTHREAD_MUTEX_INITIALIZER macro for static mutex
  - Use pthread_mutex_init() function for dynamic mutex
- Destroying a mutex
  - Use pthread_mutex_destroy() function
    - If the mutex is locked, return EBUSY
    - Hence, the mutex should be unlocked before being destroyed

Mutex

```
#include <pthread.h>

phtread_mutex_t counter_mutex = PHREAD_MUTEX_INITIALIZER;
// Static mutex declaration

void* threadRoutine(void* argumentPointer){
        …
        pthread_mutex_lock(&counter_mutex);
        …
        // Critical section with a mutex
        pthread_mutex_unlock(&counter_mutex);
        …
        return NULL;
}
```

Mutex

```
int main(){
    …

    pthread_create(threadIDs[0], NULL, threadRoutine, NULL);
    pthread_create(threadIDs[1], NULL, threadRoutine, NULL);

    pthread_join(threadIDs[0], NULL);
    pthread_join(threadIDs[1], NULL);

    pthread_mutex_unlock(&counter_mutex);
    // Should unlock before destroy
    pthread_mutex_destroy(&counter_mutex);

    return 0;
}
```

Semaphore

- Let only limited threads run a synchronized region
- Dijkstra's P and V operations
  - P(s) operation
    - Proberen ("test" in Dutch)
    - It works like [while(s==0) wait(); s--;]
  - V(s) operation
    - Verhogen ("increment" in Dutch)
    - It works like [s++;]
  - P and V operations are atomic operations

Semapore

```
…
# include <semaphore.h>
…

sem_t semaphore;
int limit = 1;

sem_init(semaphore, limit, 0);
// Initialize a semaphore
// limit: max number of threads for the synchronized region
// 0: initial value for the semaphore

sem_wait(&semaphore);
// P operation
…
// Synchronized region with a semaphore
sem_post(&semaphore);
// V operation

sem_destroy(&semaphore);
```

Problems of Multi-thread Programming

- Race condition
  - Occur because no one knows which thread run first
  - Some results can be changed by the order of thread execution
- Unintended context switching
  - Can prevent by using synchronizing
- Deadlock
  - Occur because of synchronizing
  - Wrongly designed synchronizing makes every thread wait for the wake up signal
  - The whole process is halt