

System Programming & OS 실습

Thread programming

정지현, 안석현, 김선재

Dankook University

{wlgjsjames7224, seokhyun, rlatjswo0824}@dankook.ac.kr



01

시스템 콜 : fork()



fork()

❖ fork()의 정의

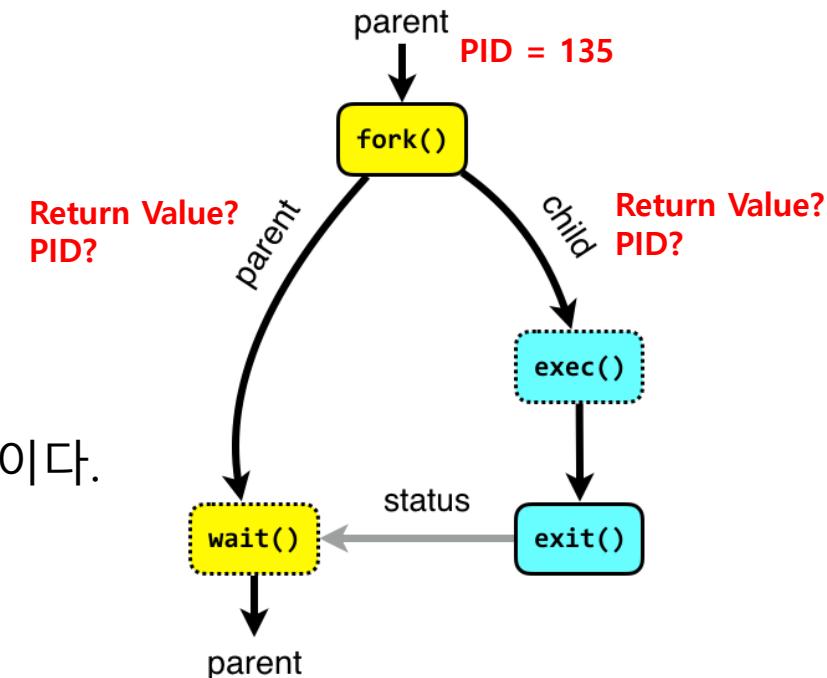
- **fork()**는 새로운 프로세스를 생성하는 시스템 호출이다.
- 생성된 새로운 프로세스는 자식 프로세스라고 하며, 부모 프로세스의 복사본이다.
- 부모와 자식 프로세스는 각각의 메모리 영역(Text, Data, Stack, Heap)을 가짐.
- 따라서 하나의 프로세스에서 발생한 변경 사항이 다른 프로세스에 영향을 주지 않음.

❖ fork()의 역할 : 제어 흐름 분할

- fork()는 프로세스를 두 개의 독립적인 제어 흐름으로 나눈다 : 부모 프로세스와 자식 프로세스
- fork() 이후, 두 개의 프로세스가 동시에 실행된다.
- 부모 프로세스는 하나이지만 부모 프로세스는 여러 개의 자식 프로세스를 가질 수 있다.

fork()

- #include <unistd.h>에 정의되어 있다.
- fork()는 부모 프로세스와 자식 프로세스에서 서로 다른 값을 반환
 - 부모 프로세스: 자식의 PID(0보다 큰 값)를 반환.
 - 자식 프로세스: 0을 반환
- 보통 자식 프로세스가 생성되면 부모 PID+1 값이 자식 프로세스 PID이다.
- PPID(Parent Process ID)는 부모 프로세스의 PID를 의미한다.
- Init 프로세스의 PID 값은 1이다.

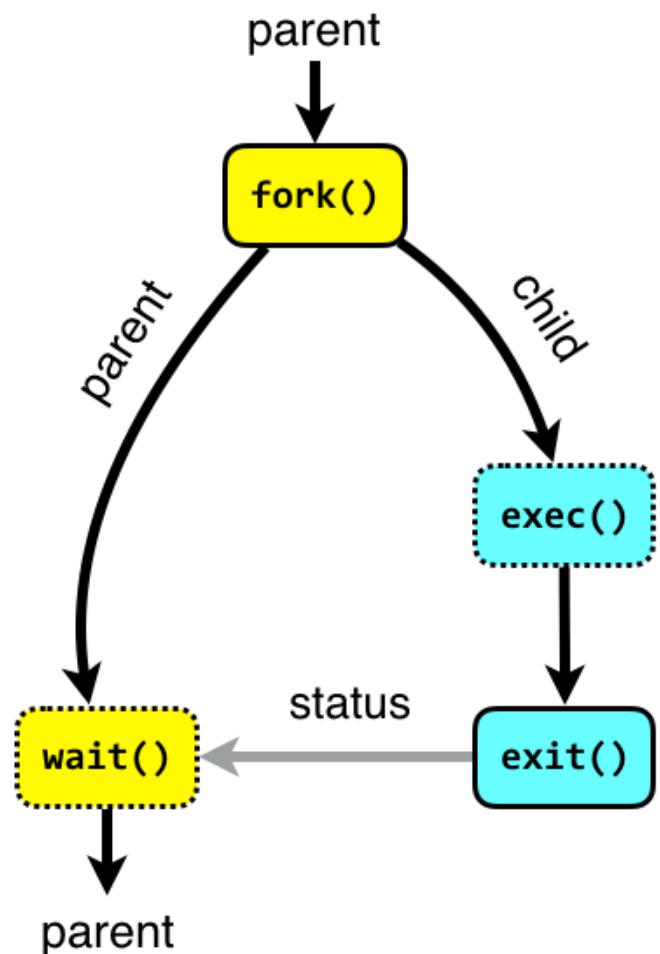


pid_t fork(void)

반환값	Value of Child	0
	Value of Parent	자식 프로세스의 PID (>0)
	Failure	-1

Related System Calls

- **fork()** : 새로운 (Child) 프로세스를 생성할 때 사용
Linux OS는 내부적으로 . . .
 - fork()가 clone()이 된다
 - task_struct를 생성한다
- **execve()** : 새로운 프로그램을 수행하고 싶을 때 로더의 역할을 수행
- **exit()** : 특정 프로세스를 종료하고 싶을 때 사용
- **wait()** : 자식 프로세스의 종료를 기다릴 때 사용
- **getpid()**: 자신의 pid를 알고 싶을때 사용



fork() 실습1

fork_test1.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
    pid_t fork_return;
    printf("Hello, my pid is %d\n", getpid());

    if ((fork_return = fork()) < 0) {
        perror("fork error");
        exit(1);
    } else if (fork_return == 0) {
        /* 자식 프로세스 */
        printf("child: pid = %d, ppid = %d\n", getpid(), getppid());
    } else {
        /* 부모 프로세스 */
        wait();
        printf("parent: I created child with pid=%d\n", fork_return);
    }

    /* 부모 프로세스와 자식 프로세스 모두 실행 */
    printf("Bye, my pid is %d\n", getpid());
}
```

- **pid_t fork_return** : fork()의 반환 값을 저장하는 변수
- **printf("Hello, my pid is %d\n", getpid())** : 부모 프로세스에서 자신의 PID를 출력.
- **fork_return = fork()** : fork()을 호출하여 자식 프로세스를 생성.
 - **fork_return < 0** : fork()가 실패했을 때 실행.
 - **fork_return == 0** : 자식 프로세스에서 실행.
 - **else** : 부모 프로세스에서 실행되며, 자식 프로세스가 종료될 때까지 wait()로 기다림.
- **printf("Bye, my pid is %d\n", getpid())** : 부모와 자식 프로세스에서 각각 한 번씩 실행.

pid_t fork(void)

반환값	Value of Child	0
	Value of Parent	자식 프로세스의 PID (>0)
	Failure	-1

fork() 실습1

❖ 실행결과

```
[ec2-user@ip-172-31-8-194 day5]$ ./fork_test1
Hello, my pid is 51679
child: pid = 51680, ppid = 51679
Bye, my pid is 51680
parent: I created child with pid=51680
Bye, my pid is 51679
[ec2-user@ip-172-31-8-194 day5]$ █
```

fork() 실습 2

fork_test2.c

```
/* borrowed from "Advanced Programming in the UNIX Env." */

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int glob = 6;
char buf[] = "a write to stdout\n";

int main(void)
{
    int var = 88;
    pid_t fork_return;

    if (write(STDOUT_FILENO, buf, sizeof(buf)) != sizeof(buf)) {
        perror("write error");
        exit(1);
    }
    printf("before fork\n"); /* we don't flush stdout */

    if ((fork_return = fork()) < 0) {
        perror("fork error");
        exit(1);
    } else if (fork_return == 0) { /* child */
        glob++;
        var++; /* modify variables */
    } else {
        sleep(2); /* parent */
    }

    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

- **glob**: 전역 변수로, 초기값은 6.
- **var**: 지역 변수로, 초기값은 88.
- 라이브러리인 `printf()`와 달리, 시스템 콜인 `write()`는 버퍼를 거치지 않으므로 즉시 출력.
- **fork_return = fork()** : `fork()`을 호출하여 자식 프로세스를 생성.
 - **fork_return < 0** : `fork()`가 실패했을 때 실행.
 - **fork_return == 0** : 전역 변수 `glob`과 지역 변수 `var`를 증가.
 - **else** : `sleep(2)`로 2초간 대기 후 출력

pid_t fork(void)

반환값	Value of Child	0
	Value of Parent	자식 프로세스의 PID (>0)
	Failure	-1

fork() 실습 2

- ❖ 실행결과 : 변수 glob과 var의 값이 서로 다르게 변경. 이를 통해 부모 프로세스와 자식 프로세스의 독립성을 확인가능하다

```
[ec2-user@ip-172-31-8-194 day5]$ ./fork_test2
a write to stdout
before fork
pid = 51692, glob = 7, var = 89
pid = 51691, glob = 6, var = 88
```

02

시스템 콜 : exec()



exec()

❖ exec()의 정의

- exec()는 새로운 프로그램을 실행하는 시스템 콜이다.
- fork는 기존에 있는 동일한 프로세스를 만들 수 있을 뿐이다.

❖ exec()의 역할 : fork한 프로세스에서 새로운 프로세스 실행

- fork한 다음에 그 프로세스에서 동일하지 않은 새로운 프로세스를 수행할 수 있도록 한다.
- 현재 프로세스의 메모리 정보들(Text, Data, Stack, Heap)을 새로운 바이너리로 교체하는 역할이다.

exec()

- **#include <unistd.h>**에 정의되어 있다.
- exec 함수군은 path나 file에 지정한 명령이나 실행파일을 실행한다.
- 사용자는 편의에 따라 다른 형태의 인터페이스 사용 가능

exec()

```
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0)  
-----  
int execv(const char *path, char *const argv[]);  
  
int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0, char*const envp[]);  
  
int execve(const char *path, char *const argv[], char *const envp[]);  
  
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);  
  
int execvp(const char *file, char *const argv[])
```

argc : 인자의 수가 저장되는 int형 변수

argv : 인자의 내용이 저장되는 포인터형 배열

envp : 환경변수 값이 들어가는 포인터형 배열

|계열 : 인자를 리스트 형태로 전달
(**execl**, **execle**, **execlp**)

v계열 : 인자를 벡터로 모아서 하나의 인자로 전달
(**execv**, **execve**, **execvp**)

exec() 실습 1

exec_test1.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    pid_t fork_return, d_pid;
    int exit_status = -1;

    if ((fork_return = fork()) == -1) {
        // fork error handling
    } else if (fork_return == 0) { /* child */
        execl("./hello", "./hello", (char *)0);
        printf("Child.. I'm here\n");
        // If exec() succeeds, the above printf() is not executed!
        exit(1);
    } else { /* parent */
        d_pid = wait(&exit_status);
        printf("Parent .. I'm here\n");
        printf("exit status of process %d is %d\n", d_pid, exit_status);
    }
}
```

- **fork_return = fork()** : fork()를 호출하여 자식 프로세스를 생성.
- **fork_return < 0** : fork()가 실패했을 때 실행.
- **fork_return == 0** : 자식 프로세스에서 ./hello 프로그램을 실행.
exec() 이후 코드는 실행되지 않음.
- **else** : 부모 프로세스는 자식 프로세스가 종료될 때까지 기다리고,
자식의 종료 상태를 exit_status에 저장.

pid_t fork(void)

반 환 값	Value of Child	0
	Value of Parent	자식 프로세스의 PID (>0)
	Failure	-1

hello.c

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
    return 0;
}
```

exec() 실습 1

- ❖ 실행결과 : hello 프로그램으로 덮어 씌어지며, 기존 프로세스의 이후 코드는 실행되지 않는다.

```
[ec2-user@ip-172-31-8-194 day5]$ ./exec_test1
Child.. I'm here
Parent .. I'm here
exit status of process 52595 is 256
```

- ❖ exec 실행X

```
[ec2-user@ip-172-31-8-194 day5]$ ./exec_test1
Hello world!
Parent .. I'm here
```

exec() 실습 2

exec_test2.c

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
    int i;
    for (i = 0; argv[i] != NULL; i++)
        printf("arg %d = %s\n", i, argv[i]);

    for (i = 0; envp[i] != NULL; i++)
        printf("envp %d = %s\n", i, envp[i]);
}
```

- **argv[]: 인수를 저장하는 문자열 배열**
- **envp[]: 환경변수를 저장하는 문자열 배열**
- **실행 시 제공된 명령줄 인수와 환경 변수를 출력**
 - for 루프문들은 NULL이 될 때까지 반복하며, 각 인덱스와 내용을 출력

exec() 실습 3

exec_test3.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <wait.h>

int main(int argc, char *argv[])
{
    pid_t fork_return, d_pid;
    int exit_status;
    char *const myenvp[] = {"sys programming", "is", "fun", (char *)0};

    if ((fork_return = fork()) == -1) {
        // fork error handling
    } else if (fork_return == 0) { /* child */
        execle("./exec_test2", "./exec_test2", "Hi", "DKU", (char *)0, myenvp);
        printf("Child... I'm here\n");
        // If execle() succeeds, this printf() is not carried out!
        exit(1);
    } else { /* parent */
        d_pid = wait(&exit_status);
        printf("exit pid = %d with status = %d\n", d_pid, WEXITSTATUS(exit_status));
    }
}
```

- **fork_return = fork()** : fork()를 호출하여 자식 프로세스를 생성.
 - **fork_return < 0** : fork()가 실패했을 때 실행.
 - **fork_return == 0** : 자식 프로세스에서 프로그램을 실행. ("Hi","DKU")와 myenvp 환경변수를 전달
- **else** : 부모 프로세스는 자식 프로세스가 종료될 때까지 기다리고, 자식의 종료 상태를 exit_status에 저장.
execle()가 성공하면 print문은 실행되지 않음

pid_t fork(void)

반환 값	Value of Child	0
	Value of Parent	자식 프로세스의 PID (>0)
	Failure	-1

exec 실습 3

❖ 실행결과 : 자식 프로세스는 실행 도중 exec_test2로 교체됨

```
[ec2-user@ip-172-31-8-194 day5]$ ./exec_test3
arg 0 = ./exec_test2
arg 1 = Hi
arg 2 = DKU
envp 0 = sys programming
envp 1 = is
envp 2 = fun
exit pid = 52620 with status = 0
```

A dark blue background featuring a dense grid of white circuit board traces and component pads, creating a technical and futuristic atmosphere.

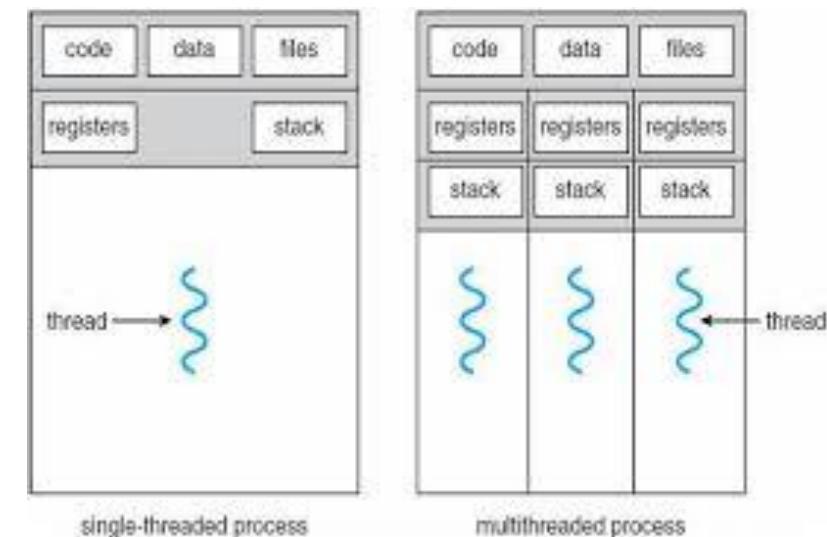
03

Single Thread Programming

Thread

❖ Thread의 정의

- CPU 사용의 기본 단위, 프로세스 내에서 병렬로 실행되는 독립적인 실행 흐름 (Execution Flow)
 - 실행흐름 : 프로그램이 어떤 방식과 순서로 실행될 것인가?
- 프로세스의 가장 작은 실행 단위, 프로그램을 실행하는 실질적인 작업자
 - 경량 프로세스라고도 불림.



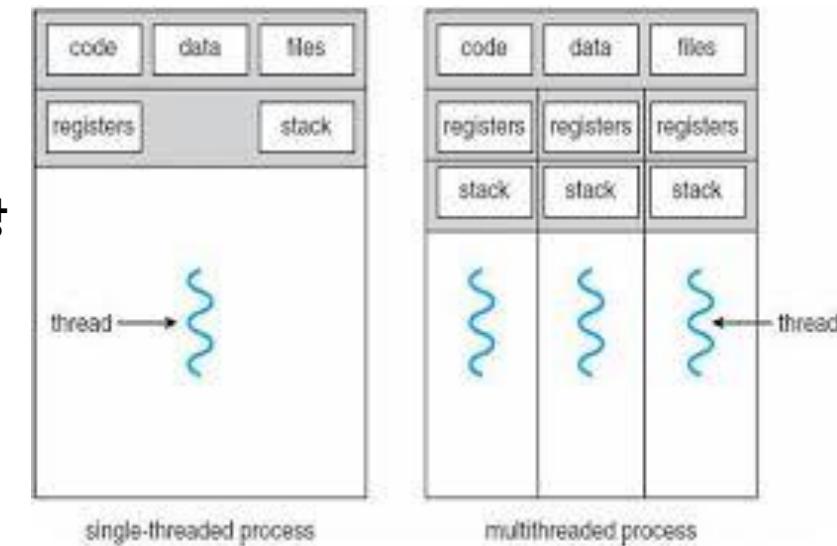
❖ 프로세스와의 비교

- 프로세스: 하나의 실행 흐름을 가지며 한 번에 하나의 작업만 수행함.
- 스레드 : 스레드를 사용하는 프로세스는 여러 개의 실행 흐름을 가질 수 있으며, 동시에 여러 작업을 수행

Thread

❖ Thread의 독립 메모리 영역

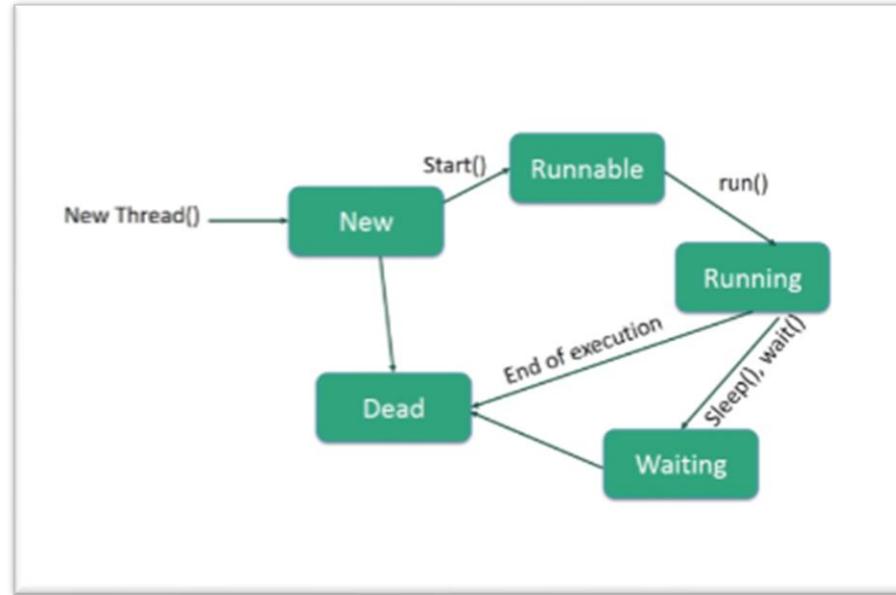
- PC (Program Counter) : 어떤 명령어를 다음으로 실행할지 나타내는 포인터
- 레지스터 집합(Register Set) : 명령어 실행 중에 필요한 데이터를 임시로 저장
- 스택(Stack) : 함수 호출 시 전달되는 인자, 반환 주소, 지역 변수를 **독립적으로** 저장



❖ Thread의 공유 메모리 영역

- 코드 영역 (Code) : 모든 쓰레드는 프로세스 내 함수나 변수등을 공유
- 데이터 영역 (Data) : 전역변수와 정적 변수가 저장. 쓰레드 간 데이터 교환 및 상태정보 공유
- 힙 영역 (Heap) : 프로세스에서 모든 쓰레드가 동적으로 메모리가 할당되고 해제
- 이외 시스템 자원 : 열린 파일 핸들, 소켓 연결등

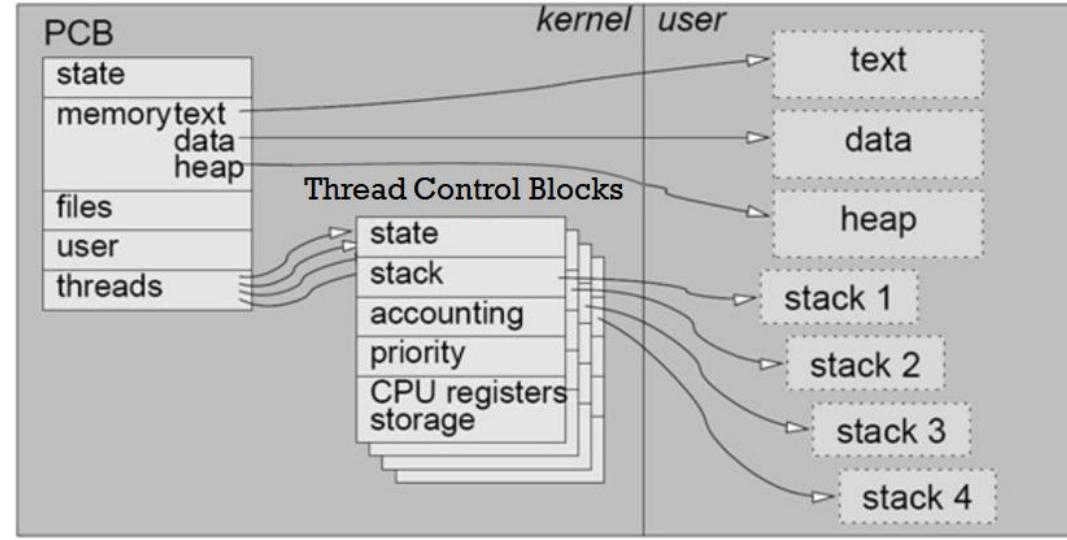
Thread 생명주기



Thread 관련 자바 메소드로 표현

- 생성(New) : 스레드가 생성되고 초기화되는 단계. 아직 실행되지 않은 상태
- 준비(Runnable) : 스레드가 실행되어, 실행 대기열에 들어가는 단계. CPU 할당을 기다림
- 실행(Running) : 스레드가 CPU를 할당받고 코드를 실행하는 단계
- 대기(Waiting) : 스레드가 특정 이벤트 (입출력 작업 완료, 다른 스레드의 신호)의 발생을 기다리는 단계. 이 때 CPU는 다른 스레드에 할당
- 종료(Dead) : 스레드의 실행이 완료되어 종료되는 단계

TCB (Thread Control Block)



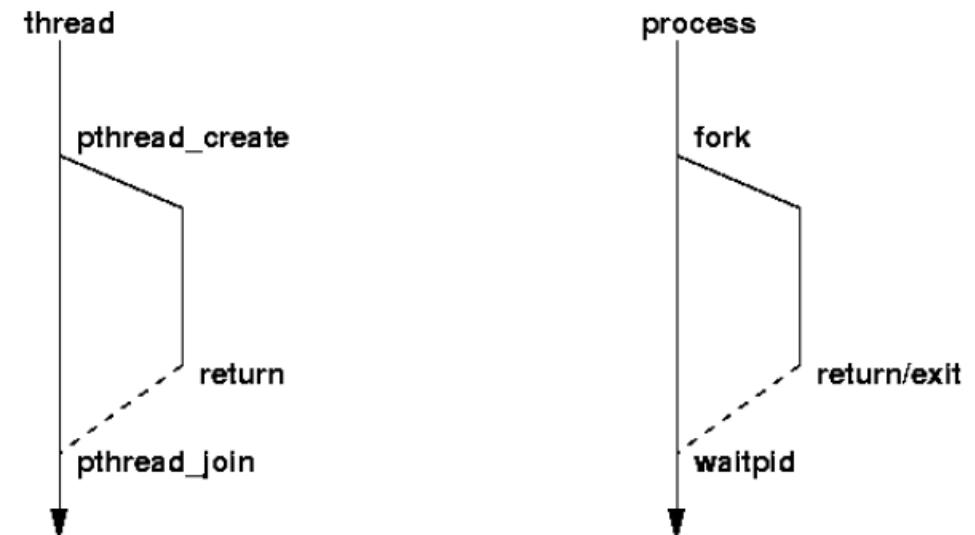
❖ TCB(Thread Control Block)는 각 스레드에 대한 정보를 저장하는 데이터 구조이며, 문맥교환을 원활하게 하기 위해 필요하다

- 스레드 식별자(Thread ID) : 시스템 내에서 각 스레드를 구별하는 고유 식별자
- 스레드 상태: 실행 중, 대기 중 등 스레드의 현재 상태.
- 스레드의 독립 영역에 관한 정보
 - 레지스터 상태: 스레드가 마지막으로 실행되었을 때의 레지스터 값.
 - 프로그램 카운터: 스레드가 다음에 실행할 명령어의 주소.
- 스택 포인터: 스레드의 스택 메모리 위치를 가리키는 포인터.

pthread

❖ pthread

- POSIX (Portable Operating System Interface) thread의 약자로, 다중 스레드 프로그래밍을 위한 표준 API
- int **pthread_create**(pthread_t * thread, const pthread_attr_t *attr, void* (*start_routine)(void*), void *arg)
 - 새로운 프로세스를 생성하는 함수
 - 인자1 : 생성된 스레드의 ID를 저장할 변수의 포인터
 - 인자2 : 스레드의 특성을 설정할 때 사용, 주로 NULL이 온다.
 - 인자3 : 스레드가 생성되고 나서 실행될 함수
 - 인자4 : 세번째 인자에서 호출되는 함수에 전달하고자 하는 인자
- int **pthread_join**(pthread_t th, void **thread_return)
 - 스레드가 종료될 때까지 호출한 스레드가 기다리도록 하게 함
 - 인자1 : 스레드 ID. 이 ID가 종료할 때까지 실행을 지연
 - 인자2 : 스레드 종료시 반환값



pthread

- void **pthread_exit**(void *retval)
 - 특정 thread에서 자신 thread를 종료한다.
 - 인자1 : void type은 모든 자료형을 포괄

❖ 이외에도...

prefix	functions
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutex routines
pthread_mutexattr_	Mutex attribute objects
pthread_cond_	Condition variable routines
pthread_condattr_	Condition attribute objects
pthread_key_	Thread_specific_data keys

<https://www.cs.fsu.edu/~baker/realtme/restricted/notes/pthreads.html>

Thread 실습1

thread_create1.c

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

void* thread_function(void* arg) {
    printf("Hello from the thread!\n");
    return NULL;
}

int main() {
    pthread_t thread; // 스레드 식별자
    int result;

    result = pthread_create(&thread, NULL, thread_function, NULL);
    if (result != 0) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
    }

    result = pthread_join(thread, NULL);
    if (result != 0) {
        fprintf(stderr, "Error joining thread\n");
        return 2;
    }

    printf("Thread has finished executing.\n");
    return 0;
}
```

- ❖ 스레드를 생성하고, 해당 스레드가 메시지를 출력한 후 종료될 때까지 대기

pthread			
반환값	pthread_create() pthread_join()	성공	0
		실패	1

Thread 실습1

- ❖ POSIX 스레드 라이브러리를 사용할 때, 컴파일 명령줄에서 **-pthread**를 추가

```
[ec2-user@ip-172-31-8-194 day5]$ gcc -pthread -o thread_test1 thread_test1.c
```

- ❖ 실행결과

```
[ec2-user@ip-172-31-8-194 day5]$ ./thread_test1
Hello from the thread!
Thread has finished executing.
```

04

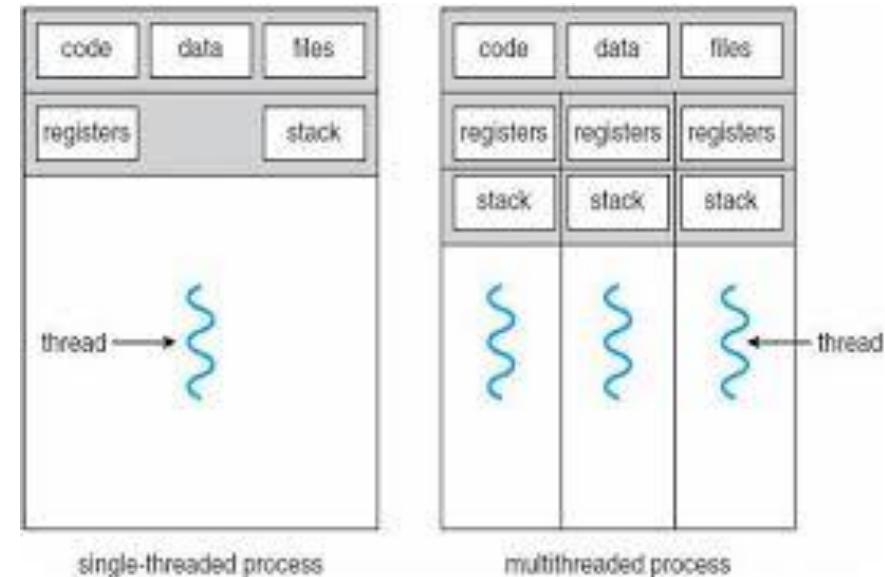
Multi Thread Programming



Multi Thread

❖ 멀티 스레드의 배경

- 스레드가 전혀 없는 프로세스는 없다
 - 이는 스레드가 경량프로세스라고 불리는 이유이다.
- 단일 스레드 프로세스는 구현이 간단하지만, 한 프로세스가 막히면 실행이 지연되고, 멀티 태스킹에 제한적이라는 단점이 있다



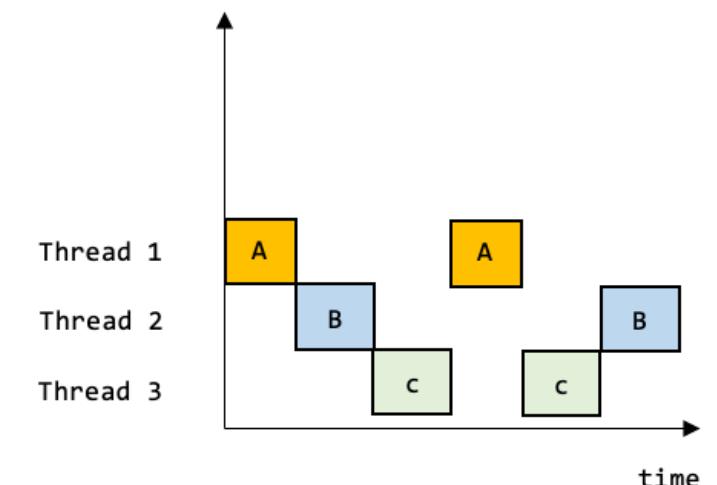
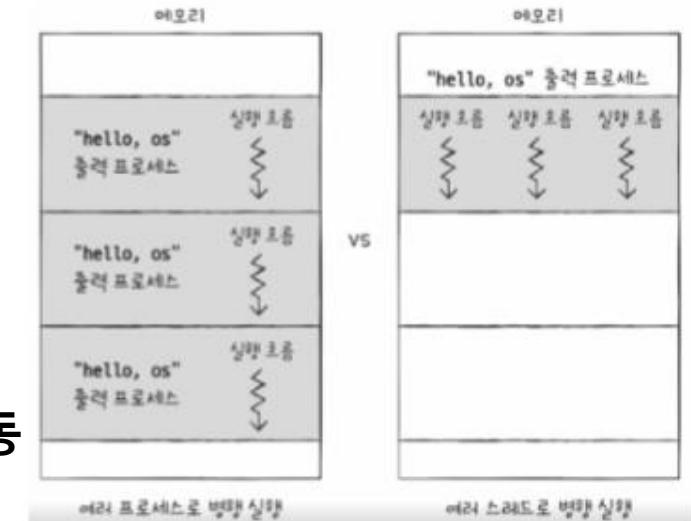
❖ 멀티 스레드 프로세스 (Multi Thread Process)

- 하나의 프로세스 내에서 여러 스레드가 동시에 실행
- 여러 작업을 동시에 수행 → 응답성이 향상되고 자원공유로 인한 효율적 사용과 처리속도 향상

Multi Thread

❖ 멀티 스레드와 멀티 프로세스의 비교

- **멀티 프로세스 :** 독립적인 메모리 공간을 가진 프로세스들이 병렬로 실행
 - 장점 : 프로세스간 독립성으로 안정성 높음
 - 단점 : 문맥교환으로 인한 오버헤드
- **멀티 스레드 :** 부모 프로세스의 자원을 공유하여 여러 실행 흐름이 동시에 작동
 - 장점 : 스레드간 자원 공유로 통신 비용이 낮음
문맥교환 비용이 프로세스에 비해 상대적으로 낮음.
 - 단점 : 하나의 스레드 오류가 전체 프로세스 실패로 이루어질 수 있음
자원 공유로 인한 동기화 문제가 발생할 수 있다.



Thread 실습2

thread_test2.c : 싱글스레드로 큰 배열의 요소를 더하기

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ARRAY_SIZE 10000000

int array[ARRAY_SIZE];

// 배열의 모든 요소를 더하는 함수
void sum_array() {
    long long sum = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += array[i];
    }
    printf("Single thread sum: %lld\n", sum);
}

int main() {
    // 배열을 초기화
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = 1; // 모든 요소를 1로 설정
    }

    clock_t start, end;
    double cpu_time_used;

    // 시간 측정 시작
    start = clock();

    // 배열의 합을 계산
    sum_array();

    // 시간 측정 종료
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Single thread execution time: %f seconds\n", cpu_time_used);

    return 0;
}
```

- 싱글스레드에서는 하나의 스레드가 전체 배열을 처리
- 멀티스레드에서는 배열을 여러 부분으로 나누어 여러 스레드가 동시에 처리
- 싱글스레드와 멀티스레드의 실행시간을 측정

Thread 실습2

thread_test3.c : 멀티스레드로 큰 배열의 요소를 더하기

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

#define ARRAY_SIZE 10000000
#define NUM_THREADS 4

int array[ARRAY_SIZE];
long long partial_sums[NUM_THREADS];

// 각 스레드에서 실행할 함수
void* sum_array(void* arg) {
    int thread_id = *((int*) arg);
    int chunk_size = ARRAY_SIZE / NUM_THREADS;
    int start = thread_id * chunk_size;
    int end = (thread_id == NUM_THREADS - 1) ? ARRAY_SIZE : start + chunk_size;

    long long sum = 0;
    for (int i = start; i < end; i++) {
        sum += array[i];
    }
    partial_sums[thread_id] = sum;
    return NULL;
}
```

```
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    long long total_sum = 0;

    // 배열을 초기화
    for (int i = 0; i < ARRAY_SIZE; i++) {
        array[i] = 1; // 모든 요소를 1로 설정
    }

    clock_t start, end;
    double cpu_time_used;

    // 시간 측정 시작
    start = clock();

    // 스레드 생성
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        if (pthread_create(&threads[i], NULL, sum_array, (void*)&thread_ids[i]) != 0) {
            perror("Failed to create thread");
            exit(1);
        }
    }

    // 모든 스레드가 종료될 때까지 대기
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // 결과 통합
    for (int i = 0; i < NUM_THREADS; i++) {
        total_sum += partial_sums[i];
    }

    // 시간 측정 종료
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Multi-threaded sum: %lld\n", total_sum);
    printf("Multi-threaded execution time: %f seconds\n", cpu_time_used);

    return 0;
}
```

Thread 실습2

❖ 실행결과

```
[ec2-user@ip-172-31-8-194 day5]$ ./thread_test2
Single-threaded sum: 100000000
Single-threaded execution time: 0.224568 seconds
[ec2-user@ip-172-31-8-194 day5]$ ./thread_test3
Multi-threaded sum: 100000000
Multi-threaded execution time: 0.379589 seconds
```

Why?

Thread 실습3

thread_test4.c : 싱글 스레드로 2개의 작업을 실행하기

```
#include <stdio.h>
#include <unistd.h> // sleep 함수 사용

int main() {
    char input[100];

    // 사용자 입력 받기
    printf("Enter a string: ");
    fgets(input, sizeof(input), stdin);

    // 사용자 입력 출력
    printf("You entered: %s", input);

    // 1부터 10까지 1초 간격으로 출력
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
        sleep(1); // 1초 동안 대기
    }

    return 0;
}
```

- **싱글 스레드** : 하나의 스레드가 입력을 받는 작업과 10~1까지 출력하는 작업을 모두 수행

Thread 실습3

thread_test5.c : 멀티 스레드로 2개의 작업을 실행하기

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h> // sleep 함수 사용

// 숫자를 1초 간격으로 출력하는 스레드 함수
void* print_numbers(void* arg) {
    for (int i = 1; i <= 10; i++) {
        printf("%d\n", i);
        sleep(1); // 1초 동안 대기
    }
    return NULL;
}

// 사용자 입력을 받는 스레드 함수
void* get_input(void* arg) {
    char input[100];

    printf("Enter a string: ");
    fgets(input, sizeof(input), stdin);
    printf("You entered: %s", input);

    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // 숫자 출력 스레드 생성
    pthread_create(&thread1, NULL, print_numbers, NULL);

    // 사용자 입력 스레드 생성
    pthread_create(&thread2, NULL, get_input, NULL);

    // 두 스레드가 종료될 때까지 대기
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

- **멀티 스레드** : 스레드1이 입력을 받고, 스레드2는 10~1까지 출력하는 작업을 수행
- **싱글 스레드와 멀티 스레드의 실행시간은 어떻게 달라질까?**

Thread 실습3

❖ 실행결과

```
[ec2-user@ip-172-31-8-194 day5]$ ./thread_test4
Enter a string: a
You entered: a
1
2
3
4
5
6
7
8
9
10
```

```
[ec2-user@ip-172-31-8-194 day5]$ ./thread_test5
1
Enter a string: a
You entered: a
2
3
4
5
6
7
8
9
10
```

- 서로 다른 자원을 사용하여 여러 개의 작업을 수행하는 경우는 멀티 스레드가 더 효율적이다.
- 싱글 스레드에서는 사용자 입력을 받을 때까지 10~1까지 출력하지 않고 기다림. 멀티 스레드의 경우 사용자가 입력을 하지 않아도 출력을 진행
- 따라서 입력을 받는 작업과 10~1까지 출력하는 두 작업이 모두 완료되기까지의 시간은 멀티 스레드를 사용했을 때 싱글 스레드보다 더 빨리 끝낼 수 있다

Thread 실습 4

none_cs.c : Critical Section을 고려하지 않음

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

static volatile int counter = 0;

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);

    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

- 2개의 스레드를 생성하고, counter에 대한 연산 수행
- 해당 소스코드에서 공유자원, 임계영역과 경쟁상태가 발생하는 시점은?

Thread 실습 4

none_cs.c : Critical Section을 고려하지 않음

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

static volatile int counter = 0;

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);

    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

```
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 12708528)
```

- $1e7 = 1\text{천만} \rightarrow$ 스레드 2개의 경우 총 2천만이어야 함
- counter 값이 2천만보다 작은 이유? \Rightarrow 경쟁상태(Race Condition)

Thread 실습 4

none_cs.c : Critical Section을 고려하지 않음

❖ Specific

High level

```
for (i = 0; i < 1e7; i++) {  
    counter = counter + 1;  
}
```



CPU level

```
100 mov    0x8049a1c, %eax  
105 add    $0x1, %eax  
108 mov    %eax, 0x8049a1c
```

- 105 line에서 다른 스레드가 끼어들면 문제 발생
 - Counter 값은 1만 증가 시켰지만, 2씩 증가된 것처럼 수행됨

Thread 실습 4

cs.c : Critical Section을 고려

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

static volatile int counter = 0;
pthread_mutex_t lock; // 뮤텍스 선언

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        pthread_mutex_lock(&lock); // 뮤텍스 잡음
        counter = counter + 1;
        pthread_mutex_unlock(&lock); // 뮤텍스 해제
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, NULL); // 뮤텍스 초기화

    printf("main: begin (counter = %d)\n", counter);

    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("main: done with both (counter = %d)\n", counter);

    pthread_mutex_destroy(&lock); // 뮤텍스 파괴
    return 0;
}
```

- **None_cs.c 와 동일한 임계 영역**
- **POSIX 라이브러리 pthread 사용**
 - **Mutex lock을 통하여 상호 배제 구현**

Thread 실습 4

cs.c : Critical Section을 고려

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

static volatile int counter = 0;
pthread_mutex_t lock; // 뮤텍스 선언

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        pthread_mutex_lock(&lock); // 뮤텍스 잡금
        counter = counter + 1;
        pthread_mutex_unlock(&lock); // 뮤텍스 해제
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    pthread_mutex_init(&lock, NULL); // 뮤텍스 초기화

    printf("main: begin (counter = %d)\n", counter);

    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    printf("main: done with both (counter = %d)\n", counter);

    pthread_mutex_destroy(&lock); // 뮤텍스 파괴
    return 0;
}
```

```
main: begin (counter = 0)
A: begin
B: begin
B: done
A: done
main: done with both (counter = 20000000)
```

- Counter 값이 본래 의도대로 2천만
- 상호 배제가 잘 지켜 졌음을 확인

Multi Thread

❖ Spin Lock

- **Spin Lock (busy waiting)** : 스레드가 임계 구역에 진입 불가 시 진입이 가능할 때 까지 CPU를 점유
 - 장점
 - 작은 작업의 경우 빠른 처리 가능
 - 구현이 간단함
 - 단점
 - CPU를 계속 점유하므로 CPU 효율성 문제 발생 가능
 - 임계 영역이 너무 길어질 경우 비효율적
 - TestAndSet, Compare-And-Swap, Fetch-And-Add, ...

Multi Thread

❖ Mutex

- **Mutex** : 스레드가 임계 구역에 진입 불가 시 Context switching 수행
 - CPU를 계속 점유하지 않고 다른 스레드에 넘기고 대기(wait), lock이 끝나면 깨어남(wake up)
 - 장점
 - 긴 임계 영역에서 유리 - CPU를 계속 점유하지 않기 때문
 - CPU의 효율적 사용 가능
 - 단점
 - Sleep 시에 Context switch 비용 발생 (expensive)
 - 특히 짧은 임계 영역에서 불리

Thread 실습 5

mutex.c : Mutex Lock 구현

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

pthread_mutex_t mutex;
int counter = 0;

// 일정 영역
void *critical_section(void *arg) {
    for (int i = 0; i < 1000; i++) {
        pthread_mutex_lock(&mutex); // Mutex 잡음
        counter++;
        pthread_mutex_unlock(&mutex); // Mutex 해제
    }
    return NULL;
}
```

```
int main() {
    pthread_t threads[2];
    struct timeval start, end;

    // Mutex 초기화
    pthread_mutex_init(&mutex, NULL);

    // 시간 측정 시작
    gettimeofday(&start, NULL);

    // 스레드 생성
    for (int i = 0; i < 2; i++) {
        pthread_create(&threads[i], NULL, critical_section, NULL);
    }

    // 스레드 종료 대기
    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    // 시간 측정 종료
    gettimeofday(&end, NULL);

    // 결과 출력
    long seconds = (end.tv_sec - start.tv_sec);
    long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);

    printf("Mutex lock completed in %ld microseconds\n", micros);
    printf("Final counter value: %d\n", counter);

    // Mutex 제거
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

- **Mutex lock**
- **실행시간 비교를 위해 #include <sys/time.h> 사용**

Thread 실습 5

spin.c : Spin Lock 구현

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

typedef struct __lock_t {
    int flag;
} lock_t;

int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr;
    *old_ptr = new;
    return old;
}

void init(lock_t *lock) {
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // Spin
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}

int counter = 0;
lock_t spinlock;

// 임계 영역
void *critical_section(void *arg) {
    for (int i = 0; i < 1000; i++) {
        lock(&spinlock);
        counter++;
        unlock(&spinlock);
    }
    return NULL;
}
```

```
int main() {
    pthread_t threads[2];
    struct timeval start, end;

    init(&spinlock);

    // 시간 측정 시작
    gettimeofday(&start, NULL);

    // 스레드 생성
    for (int i = 0; i < 2; i++) {
        pthread_create(&threads[i], NULL, critical_section, NULL);
    }

    // 스레드 종료 대기
    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    // 시간 측정 종료
    gettimeofday(&end, NULL);

    // 결과 출력
    long seconds = (end.tv_sec - start.tv_sec);
    long micros = ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);

    printf("Spin lock completed in %ld microseconds\n", micros);
    printf("Final counter value: %d\n", counter);

    return 0;
}
```

- Spin lock

- Test-And-Set (Atomic Exchange) 기법 사용

- Test-And-Set을 사용해서 원자성 보장

Thread 실습 5

Mutex와 Spin(Test-And-Set) 실행 결과 비교

Mutex

```
./mutex
Mutex lock completed in 814 microseconds
Final counter value: 2000
```

Spin

```
./spin
Spin lock completed in 746 microseconds
Final counter value: 2000
```

- 실행 시간의 경우 유의미한 차이 X
 - 임계영역이 짧고 단순한 구조이기 때문
 - Counter 값은 모두 2000번으로 정상적으로 프로그램이 동작함을 확인

05

System Call in Thread Programming



fork() in Thread

❖ 단일 스레드 프로세스(=프로세스)에서의 fork() 호출

- 프로세스 공간의 모든 데이터가 새 프로세스에 복사되어 생성됨

❖ 멀티 스레드 프로세스의 fork() 호출

- 1. 모든 스레드 복사 : 부모 프로세스에서의 사용중인 모든 스레드를 새 프로세스에 복사함
 - 서버의 각 스레드가 **독립적으로 클라이언트 요청을 처리하고 있다** → 요청 처리의 일관성을 위해 그대로 복사하는 것이 낫다
- 2. 호출한 스레드만 복사 : fork()를 호출한 단일 스레드만 새 프로세스로 복사함 → 스택영역만 복사
 - 새로운 서비스나 작업을 시작할 때, **불필요한 스레드를 복사하지 않는 것이 오버헤드를 줄임**

exec() in Thread

❖ 단일 스레드 프로세스(=프로세스)에서의 exec() 호출

- PID는 유지
- 실행중이던 프로그램은 종료되고, 지정된 새 프로그램의 실행이 시작됨

❖ 멀티 스레드 프로세스의 fork() 호출

- PID는 유지
- 모든 스레드는 종료되며, 프로세스는 새로운 프로그램을 실행하는 단일 스레드 프로세스로 변환됨.

exec() after fork() in Thread

❖ fork() 후 exec() 호출

- fork()시 부모 프로세스의 스레드들을 어떻게 처리해야 할까?
 - Case 1 : 부모 프로세스의 모든 스레드를 복사 → exec()를 호출하여 새로운 프로그램으로 대체한다
 - Case 2 : 호출한 단일 스레드만 복사 → 호출한 스레드만 복사한 후, exec()를 호출하여 새로운 프로그램으로 대체함
- 대부분의 경우, Case2가 더 적합하고 선호되는 설계 방식이다
 - fork()후 exec()를 사용하는 일반적인 패턴은, 새 프로세스를 최소한의 상태로 시작하여 exec() 호출로 새로운 프로그램을 효율적으로 시작하는 것

Thread 실습4

thread_test6.c

```
#include <ptnread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NUM_THREADS 4

// 스레드가 수행할 작업
void* thread_function(void* arg) {
    pthread_t thread_id = pthread_self();
    printf("Thread %lu is running before exec\n", thread_id);

    // 스레드가 잠시 대기
    sleep(1);

    // exec 호출 전에 현재 스레드와 다른 스레드가 모두 종료됨
    // exec 호출 후에는 새로운 프로그램이 단일 스레드로 실행됨
    execlp("date", "date", NULL); // 'date' 명령어로 현재 날짜와 시간을 출력
    perror("execlp failed"); // execlp가 실패하면 여기로 오게됨

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];

    // 스레드 생성
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, thread_function, NULL) != 0) {
            perror("Failed to create thread");
            exit(1);
        }
    }

    // 모든 스레드가 종료될 때까지 대기
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    // exec 호출 후, 모든 스레드는 종료되고 새로운 프로그램이 실행됨
    // 이 위치의 코드는 이론적으로 도달하지 않음
    printf("This line will not be reached if exec() is successful\n");

    return 0;
}
```

- **fork() 후, 멀티 스레드 환경에서 exec()를 호출하면, 현재 프로세스의 모든 스레드를 종료시키고 새로운 프로그램으로 대체**
 - 프로세스는 새로운 프로그램의 단일 스레드로 동작하며, 기존의 모든 스레드는 종료.
- execlp("date", "date", NULL);는 현재 프로세스를 date 명령어로 대체.

pthread

반환값	pthread_create() pthread_join()	성공	0
		실패	1

스케줄링 실습

❖ 실행결과

```
[ec2-user@ip-172-31-8-194 day5]$ ./thread_test6
Thread 139929508116032 is running before exec
Thread 139929499723328 is running before exec
Thread 139929491330624 is running before exec
Thread 139929482937920 is running before exec
Tue Aug 27 02:17:19 AM UTC 2024
```

- 단일 스레드 프로세스에서 fork() 후, 4개의 스레드가 생성되었을 때 exec()를 호출하면, 현재 프로세스의 모든 스레드를 종료시키고 현재 시간을 출력하는 프로그램으로 대체
 - 프로세스는 새로운 프로그램의 단일 스레드로 동작

A dark blue background featuring a dense grid of white circuit board traces and component pads, creating a technical and futuristic feel.

06

Scheduling

스케줄링의 필요성

❖ 준비 큐에 있는 프로세스 중 어떤 것부터 CPU에 할당할까 ?

- 목표 프로세스의 실행 순서를 결정 → CPU 이용률, 처리량, 응답시간 등을 최적화 하는 것
- 처리 코어의 가정 : 단일 코어로 가정함 → 실제 시스템에서는 여러 코어에 걸쳐 확장될 수 있음

❖ 주요 평가 기준

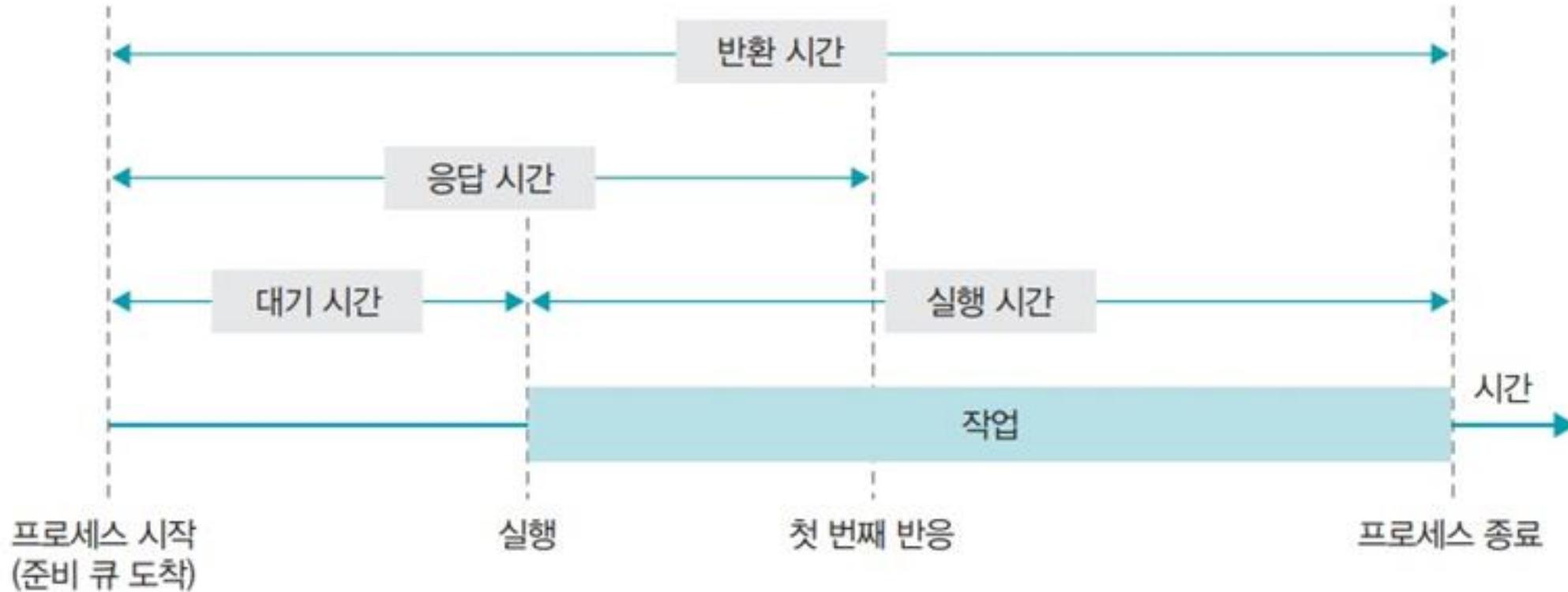
▪ 최대화해야 하는 기준

- **CPU Utilization**(CPU 사용률) : CPU는 비싼 자원이기에 바쁘면 바쁠수록 좋음.
- **Throughput**: 단위시간(time unit)당 작업수행을 완료한 프로세스의 수.

▪ 최소화해야 하는 기준

- **평균 대기 시간(Average Waiting Time)** : 프로세스가 준비 큐에서 실행을 기다리는 데 소요된 시간의 평균
 - 스케줄링 알고리즘의 중요한 목표 → 평균 대기시간 최소화하기
- **대화식 시스템에서의 응답시간 (Response Time)** : 연구에 따르면 응답시간의 최소화보다 응답시간의 편차를 최소화 하는 것이 사용자 경험 향상에 중요
 - 단순화를 위해 주로 CPU 버스트 시간만을 고려하지만, 실제 상황에서는 CPU 버스트와 I/O 버스트도 고려될 수 있음

스케줄링 진행



- **도착시간 : 준비 큐까지 프로세스가 도착하는 시간**
 - 프로세스가 실행 준비를 마치고, 준비 큐에 추가된 시점을 나타낸다

Preemptive, Non-Preemptive Scheduling

- **비선점형 스케줄링 (Non-Preemptive Scheduling)**

- 한 번 CPU가 프로세스에 할당되면, 그 프로세스가 스스로 실행을 완료하거나, I/O 요청 등으로 CPU가 방출될 때까지 CPU를 점유
- 실시간 시스템에서는 제한적이거나 부적합할 수 있음.
- FCFS, SJF, Priority 등

- **선점형 스케줄링 (Preemptive Scheduling)**

- 스케줄러가 현재 CPU를 점유하고 있는 프로세스를 강제로 중단시키고, 다른 프로세스에게 CPU를 할당하는 방식
- 멀티 태스킹 환경에서 효율적인 자원 분배를 가능하게 함 → 대부분의 최신 운영체제에서 택함.
- 하지만 여러 프로세스나 스레드가 동시에 커널 데이터 구조에 접근할 수 있기 때문에 경쟁 상태(Race Condition) 발생 가능
- RR, SRTF, Priority, MLQ, **MLFQ** 등

❖ FCFS : First-Come-First-Served

- 비선점형 스케줄링 방식
- 특징
 - 구현의 단순성 : 구현과 관리가 매우 간단하여 초기 컴퓨터 시스템에 사용
 - 공정성 : 모든 프로세스가 도착한 순서대로 처리
- 한계
 - 응답시간 : 한 프로세스가 CPU를 지나치게 점유해 다른 프로세스의 응답시간이 길어질 수 있음 → 컨벡스 효과(Convey Effect)
 - 대화형 시스템에서 부적합 : FCFS는 규칙적으로 CPU 시간을 할당하기 어려워 사용자 입력에 대한 즉각적인 응답이 자연될 수 있음

Non-Preemptive SJF

❖ Non-Preemptive SJF : Shortest Job First

- 비선점형 스케줄링 방식
- CPU 버스트 시간 기준 할당: 준비 큐에 있는 프로세스 중 예상 CPU 버스트 시간이 가장 짧은 것을 우선적으로 CPU를 할당받음
- 특징
 - 최소화된 평균 대기 시간 : 이론적으로 최단 작업을 먼저 처리함으로써 평균 대기 시간을 최소화할 수 있음
 - 간단한 예측 모델 : 프로세스의 다음 CPU 버스트 시간을 예측하기 위해, 과거의 버스트 시간의 평균을 계산함
- 한계
 - 기아 현상 : 짧은 작업이 계속 도착하면 장시간 동안 실행되야하는 프로세스는 CPU를 할당 받지 못할 수 있음
 - CPU 버스트 시간의 예측 매우 어려움 → 잘못된 예측으로 스케줄링의 효율성을 크게 떨어뜨릴 수 있음
 - 실시간 처리에 부적합 : 비선점형이고, 예측된 처리 시간이 짧은 작업에 유리하기 때문에, 긴급한 처리가 필요한 실시간 시스템에서는 적합하지 않을 수 있음

Priority

❖ Priority

- 비선점형/선점형 스케줄링 방식
- 우선순위 할당 : 각 프로세스는 생성시 고유의 우선순위를 정적으로, 혹은 동적으로 부여받음
- 특징
 - 유연성 : 다양한 우선순위를 가진 작업을 효율적으로 유연하게 처리
 - 응답 시간 개선 : 긴급한 작업에 높은 우선순위를 부여하여 빠른 응답 시간을 보장할 수 있음
 - 적응성 : 동적 우선순위 하당을 통해 시스템의 변화에 적응하며 최적의 성능을 유지할 수 있음
- 한계
 - 무한봉쇄 / 기아 상태 : 우선 순위가 낮은 프로세스가 높은 우선순위의 프로세스들로 인해 무한히 대기하는 현상

Round Robin

❖ RR : Round Robin

- 선점형 스케줄링 알고리즘
- 시간 할당량 부여 : 각 프로세스는 CPU를 사용할 수 있는 고정된 시간(Time slice, 10~100ms)을 할당 받음
- 특징
 - 원형 큐 관리 : 준비 큐는 원형 큐 방식으로 관리되며 프로세스는 FIFO순서로 CPU 시간을 할당 받음
 - 타이머 인터럽트 : 각 프로세스에 할당 시간이 끝나면 타이머 인터럽트가 발생하고, CPU는 다음 프로세스로 전환
- 프로세스 전환 시나리오
 - 1. CPU 버스트 < 시간 할당량 : 이미 프로세스가 끝나버리면 CPU를 방출하고 준비 큐의 다음 프로세스가 CPU를 할당 받음
 - 2. CPU 버스트 > 시간 할당량 : 타이머가 종료되면 운영체제는 인터럽트를 발생시켜 현재 프로세스를 중단하고 큐의 끝으로 이동시킴
- 성능 영향 요인
 - 너무 큰 시간 할당량 : 사실 FCFS와 동일하게 작동함
 - 너무 작은 시간 할당량 : 빈번한 컨텍스트 스위칭으로 인한 오버헤드 증가

SRTF : Shortest Remaining Time First

- 선점형 스케줄링 알고리즘
- SJF 알고리즘을 비선점형에서 선점형으로 변경 (선점형 SJF)
 - 특징과 한계점도 SJF와 동일

❖ MLQ : Multi Level Queue

- 선점형 스케줄링 알고리즘
- 준비 큐를 다양하게 준비 → 프로세스의 특성에 따라 다른 큐에 할당됨
- 주요 큐 예시
 - 스케줄러는 높은 우선순위의 큐부터 순차적으로 프로세스를 실행시키며, 같은 우선순위 내에서는 해당 큐의 스케줄링 정책에 따라 프로세스가 관리
 - 1. 실시간 프로세스 큐 : 가장 높은 우선순위, 시간제약이 중요한 작업을 처리
 - 2. 시스템 프로세스 큐 : 핵심 시스템 작업
 - 3. 대화형 프로세스 큐 : 사용자와 직접 상호작용하는 프로세스를 포함하며 신속한 응답시간 요구
 - 4. 배치 프로세스 큐 : 가장 낮은 우선순위 → 자동화된 작업을 처리함
- 한계
 - 융통성 부족 : 프로세스가 한 번 특정 큐에 할당되면, 그 큐 내에서만 스케줄링 → 다른 큐로의 이동이 제한되어 융통성이 떨어질 수 있음
 - 기아 상태 : 낮은 우선순위의 큐에 있는 프로세스는 CPU할당을 받지 못하고 기다려야 하는 기아 상태에 빠질 위험이 있음
 - 우선순위 역전 문제 : 낮은 우선순위의 프로세스가 고유의 자원을 점유하고 있을 때, 높은 우선순위의 프로세스가 해당 자원을 기다려야 하는 상황이 발생할 수 있음

❖ MLQ : Multi Level Feedback Queue

- 다단계 큐 스케줄링은 다른 큐 이동 X → 하지만 MLFQ는 성능 피드백을 바탕으로 큐를 동적으로
- 큐의 개수와 스케줄링 알고리즘 : 시스템 내에는 여러 개의 큐가 존재하며, 각 큐마다 다른 스케줄링 알고리즘(RR, Priority등)이 적용
- **프로세스 관리**
 - 큐 선택 : 새로운 프로세스나 준비 상태로 돌아오는 프로세스가 처음에 들어갈 큐를 결정
 - 우선순위 증가 : 오랜 시간 대기한 프로세스나 특정 조건을 만족하는 프로세스의 우선순위를 높임
 - 우선순위 감소 : 너무 자주 CPU를 사용하는 프로세스나 시스템 자원을 과도하게 사용하는 프로세스의 우선순위를 낮춤
- **특징**
 - 유연성 : 프로세스의 우선순위와 큐 위치는 실행중인 프로세스의 행동과 시스템의 상태에 따라 유동적으로 저정됨
 - 기아상태 방지 : 노화 메커니즘을 통해 오랜 시간동안 대기하는 프로세스의 우선순위를 점차적으로 증가시켜 모든 프로세스가 실행될 기회를 보장
 - 공정성 : 시스템은 다양한 종류의 프로세스가 요구 사항을 충족시키면서도 모든 프로세스에 공정한 실행 기회를 제공함
- **한계**
 - 복잡성 : 다양한 큐와 우선순위 규칙, 프로세스간 이동 로직은 시스템 설계가 어려움 : 오버헤드 증가, 적절한 파라미터 설정의 어려움

스케줄링 실습(RR)

round_robin.c

```
#include <stdio.h>

#define MAX_PROCESSES 10

typedef struct {
    int id;
    int burst_time;
    int remaining_time;
} Process;

void round_robin(Process processes[], int n, int time_quantum) {
    int time = 0;
    int processes_left = n;

    while (processes_left > 0) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                int run_time = (processes[i].remaining_time > time_quantum)
                    ? time_quantum : processes[i].remaining_time;

                printf("Time %d - %d: Process %d is running for %d units\n",
                    time, time + run_time, processes[i].id, run_time);

                time += run_time;
                processes[i].remaining_time -= run_time;

                if (processes[i].remaining_time == 0) {
                    processes_left--;
                }
            }
        }

        printf("All processes are completed.\n");
    }
}
```

- Process 구조체에서는 PID, 버스트 시간, 남은 실행 시간(Remaining_time)을 정의
- round_robin 함수
 - 프로세스의 남은 시간 \leq time_quantum : 전체 남은 시간을 실행
 - 프로세스의 남은 시간 $>$ time_quantum : time_quantum 만큼 실행

스케줄링 실습(RR)

round_robin.c

```
int main() {
    int n, time_quantum;

    printf("Enter the number of processes (max %d): ", MAX_PROCESSES);
    scanf("%d", &n);

    if (n > MAX_PROCESSES) {
        printf("Number of processes cannot exceed %d\n", MAX_PROCESSES);
        return 1;
    }

    Process processes[MAX_PROCESSES];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }

    printf("Enter time quantum: ");
    scanf("%d", &time_quantum);

    round_robin(processes, n, time_quantum);

    return 0;
}
```

- 간단한 Round Robin 스케줄러 구현

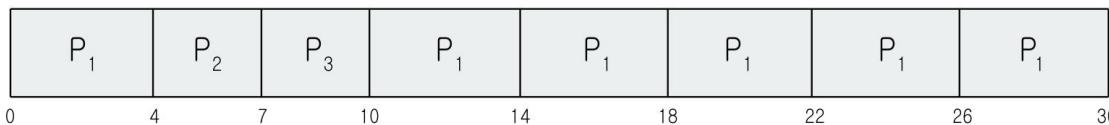
- 프로세스 수와 각 프로세스의 실행 시간, 시간 슬라이스를 입력받고, round robin 함수를 호출하여 스케줄링을 수행

스케줄링 실습

❖ 실행결과

- 모든 프로세스는 도착시간은 0으로 가정

Process	Arrival Time	Burst Time
P ₁	0	24
P ₂	0	3
P ₃	0	3



```
[ec2-user@ip-172-31-8-194 day5]$ ./round_robin
Enter the number of processes (max 10): 3
Enter burst time for process 1: 24
Enter burst time for process 2: 3
Enter burst time for process 3: 3
Enter time quantum: 4
Time 0 - 4: Process 1 is running for 4 units
Time 4 - 7: Process 2 is running for 3 units
Time 7 - 10: Process 3 is running for 3 units
Time 10 - 14: Process 1 is running for 4 units
Time 14 - 18: Process 1 is running for 4 units
Time 18 - 22: Process 1 is running for 4 units
Time 22 - 26: Process 1 is running for 4 units
Time 26 - 30: Process 1 is running for 4 units
All processes are completed.
```