**multi-process 구현 학습**운영체제 02분반

컴퓨터공학과

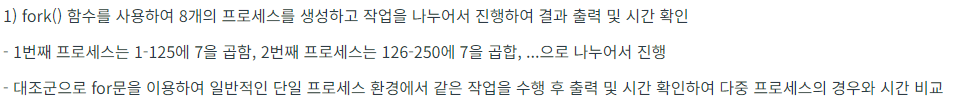
2142851 김형준

**fork() 함수를 이용한 프로그램 작성**

**목차**

* **8개의 프로세스**
  1. 코드
  2. 결과 출력
  3. 시간 비교
* **10개의 프로세스**
  1. 코드
  2. 결과 출력
  3. 시간 비교
  4. 추가 실험
* **기능 추가 코드**

1. **8개의 프로세스**



1. **코드**

코드의 설명은 주석처리로 하였습니다.

(단일 프로세스 제외)

단일 프로세스 (8-process-single.cpp)

*#include* <stdio.h>

*#include* <time.h>

int main() {

    double totalSum = 0;

    int repeat = 10;

*for*(int testCase = 0; testCase < repeat; testCase++) {

        clock\_t startTime, endTime;

        double totalTime;

        startTime = clock();

*for* (int i = 1; i <= 1000; ++i) {

            printf("%d \* %d = %d\n", i, 7, i \* 7);

        }

        endTime = clock();

        totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

        printf("Total Time: %f seconds\n", totalTime);

        totalSum += totalTime;

    }

    printf("[Average time after %d repetitions: %f]\n", repeat, totalSum / repeat);

*return* 0;

}

다중 프로세스 (8-process-fork.cpp)

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <sys/types.h>

*#include* <unistd.h>

*#include* <sys/wait.h>

*#include* <time.h>

*// process가 i\*7 연산을 하기 위한 함수*

void process\_task(int *start*, int *end*, int *process\_num*) {

*// printf("Process %d - Start: %d, End: %d\n", process\_num, start, end);*

*// 각 프로세스에서의 시간도 측정해보기 위해 시간 측정 알고리즘 추가*

    clock\_t startTime = clock();

*// 연산*

*for* (int i = start; i <= end; ++i) {

        printf("%d \* %d = %d\n", i, 7, i \* 7);

    }

    clock\_t endTime = clock();

    double totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

*// 아래 printf는 시간 측정 시 주석 처리하였습니다.*

    printf("Process %d end\n", process\_num);

*// printf("Process %d Elapsed Time: %f seconds\n", process\_num, totalTime);*

*// printf("---------------------------------------------\n");*

}

int main() {

*// 반복 후 평균을 내기 위해 repeat회 결과값을 더하고 저장할 변수*

    double totalSum = 0;

*// 반복 횟수*

    int repeat = 10;

*// repeat만큼 반복*

*for*(int testCase = 0; testCase < repeat; testCase++) {

*// startTime: 시작 시간*

*// endTime: 끝 시간*

        clock\_t startTime, endTime;

*// totalTime: startTime과 endTime을 연산 후 나온 결과 (실행 시간)*

        double totalTime;

        startTime = clock(); *// 시작 시간 기록*

*// pid: 프로세스 ID 저장*

        pid\_t pid;

*// start: 연산할 값의 시작 값*

*// end: 연산할 값의 마지막 값 (+ 1)*

*// range: 각 프로세스의 처리 범위*

        int start, end, range, i;

        range = 1000 / 8;

*// 프로세스의 개수만큼 반복*

*for* (i = 0; i < 8; ++i) {

            start = i \* range + 1;

            end = (i + 1) \* range;

*// 자식 프로세스 생성*

            pid = fork();

*if* (pid < 0) { *// fork 실패*

                perror("fork");

                exit(1);

            } *else* *if* (pid == 0) { *// 자식 프로세스*

*// process\_task 함수로 이동*

*// process\_task(시작 값, 마지막 값, 프로세스 번호)*

                process\_task(start, end, i);

                exit(0);

            }

        }

*// 자식 프로세스 종료 대기*

*for* (i = 0; i < 8; ++i) {

            wait(NULL);

        }

        endTime = clock(); *// 종료 시간*

        totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC; *// 경과 시간*

        printf("Total Time: %f seconds\n", totalTime);

        totalSum += totalTime;

    }

*// 평균 시간 계산 후 출력*

    printf("[Average time after %d repetitions: %f]\n", repeat, totalSum / repeat);

*return* 0;

}

1. **결과 출력**

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

1. **시간 비교**

|  |  |
| --- | --- |
| **단일 프로세스** | **다중 프로세스** |
| 0.002391 seconds | 0.000652 seconds |

(1,000까지의 연산)

**프로세스를 8개**로 돌린다고 **시간이 8배 단축되는 것은 아니었습니다.**

**그 이유로는** 다음을 추측해볼 수 있었습니다.

- 프로세스를 생성하고 종료할 때 **오버헤드 발생**으로 인해 시간이 발생합니다.

- 곱하기 7의 경우 **간단한 CPU 연산**이므로, 시간 감소의 크기가 제한될 수 있습니다.

- **CPU 사용 순서를 결정함**에 있어, 아주 짧지만 추가적인 시간이 필요합니다.

이를 통해, 프로세스를 무작정 늘리는 것이 성능을 향상시키기 위한 좋은 방법은 아니라고 생각했습니다.

추가로, 1,000개의 숫자가 아닌 100,000개의 숫자를 연산해보았습니다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

|  |  |
| --- | --- |
| **단일 프로세스** | **다중 프로세스** |
| 0.171066 seconds | 0.000425 seconds |

(100,000까지의 연산) + (10회 평균)

예상과는 다르게**, 8배보다 아주 많이 단축된 모습**이었습니다.

심지어, 1,000까지의 연산보다 시간이 더 단축된 모습을 볼 수 있었습니다.

이 이유에 대해 Chat GPT에게 물어보았습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

이후 코드를 수정하여, 10번 반복 후 평균을 낼 수 있도록 하였습니다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

추가한 코드는 다음과 같습니다. (int main 부분만 복사하였습니다.)

(1번의 코드의 일부분입니다.)

int main() {

    double totalSum = 0;

    int repeat = 10;

*for*(int testCase = 0; testCase < repeat; testCase++) {

        ...

        totalSum += totalTime;

    }

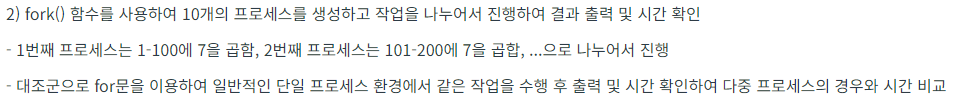
    printf("[Average time after %d repetitions: %f\n]", repeat, totalSum / repeat);

*return* 0;

}

test case를 늘려, 더욱 신뢰성 있는 결과를 도출하도록 노력했습니다.

1. **10개의 프로세스**



1. **코드**

단일 프로세스 (10-process-single.cpp)

(위 코드와 동일합니다.)

다중 프로세스 (10-process-fork.cpp)

(다중 프로세스 8개보다 유지보수에 신경을 썼습니다. 큰 차이는 없습니다.)

*#include* <stdio.h>

*#include* <stdlib.h>

*#include* <sys/types.h>

*#include* <unistd.h>

*#include* <sys/wait.h>

*#include* <time.h>

*// 프로세스의 개수*

*#define* PROCESSCOUNT 10

*#define* MULVALUE 7

*// process가 i\*MULVALUE 연산을 하기 위한 함수*

void process\_task(int *start*, int *end*, int *process\_num*) {

*// printf("Process %d - Start: %d, End: %d\n", process\_num, start, end);*

*// 각 프로세스에서의 시간도 측정해보기 위해 시간 측정 알고리즘 추가*

    clock\_t startTime = clock();

*// 연산*

*for* (int i = start; i <= end; ++i) {

        printf("%d \* %d = %d\n", i, MULVALUE, i \* MULVALUE);

    }

    clock\_t endTime = clock();

    double totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

*// 아래 printf는 시간 측정 시 주석 처리하였습니다.*

    printf("Process PID:%d\n", (int)getppid()); *// Process PID Check*

    printf("Process %d end\n", process\_num);

*// printf("Process %d Elapsed Time: %f seconds\n", process\_num, totalTime);*

*// printf("---------------------------------------------\n");*

}

int main() {

*// 반복 후 평균을 내기 위해 repeat회 결과값을 더하고 저장할 변수*

    double totalSum = 0;

*// 반복 횟수*

    int repeat = 10;

*// repeat만큼 반복*

*for*(int testCase = 0; testCase < repeat; testCase++) {

*// startTime: 시작 시간*

*// endTime: 끝 시간*

        clock\_t startTime, endTime;

*// totalTime: startTime과 endTime을 연산 후 나온 결과 (실행 시간)*

        double totalTime;

        startTime = clock(); *// 시작 시간 기록*

*// pid: 프로세스 ID 저장*

        pid\_t pid;

*// start: 연산할 값의 시작 값*

*// end: 연산할 값의 마지막 값 (+ 1)*

*// range: 각 프로세스의 처리 범위*

        int start, end, range, i;

        range = 1000 / PROCESSCOUNT;

*// 프로세스의 개수만큼 반복*

*for* (i = 0; i < PROCESSCOUNT; ++i) {

            start = i \* range + 1;

            end = (i + 1) \* range;

*// 자식 프로세스 생성*

            pid = fork();

*if* (pid < 0) { *// fork 실패*

                perror("fork");

                exit(1);

            } *else* *if* (pid == 0) { *// 자식 프로세스*

*// process\_task 함수로 이동*

*// process\_task(시작 값, 마지막 값, 프로세스 번호)*

                process\_task(start, end, i);

                exit(0);

            }

        }

*// 자식 프로세스 종료 대기*

*for* (i = 0; i < PROCESSCOUNT; ++i) {

            wait(NULL);

        }

        endTime = clock(); *// 종료 시간*

        totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC; *// 경과 시간*

        printf("Total Time: %f seconds\n", totalTime);

        totalSum += totalTime;

    }

*// 평균 시간 계산 후 출력*

    printf("[Average time after %d repetitions: %f]\n", repeat, totalSum / repeat);

*return* 0;

}

1. **결과 출력**

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

1. **시간 비교**

1회차

|  |  |
| --- | --- |
| **단일 프로세스** | **다중 프로세스** |
| 0.001471 seconds | 0.000568 seconds |

(10회 반복 후 평균)

2회차

|  |  |
| --- | --- |
| **단일 프로세스** | **다중 프로세스** |
| 0.001998 seconds | 0.000916 seconds |

(10회 반복 후 평균)

3회차

|  |  |
| --- | --- |
| **단일 프로세스** | **다중 프로세스** |
| 0.001757 seconds | 0.000538 seconds |

(10회 반복 후 평균)

4회차 (단일 프로세스, 8개 프로세스, 10개 프로세스)

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

|  |  |  |
| --- | --- | --- |
| **단일 프로세스** | **다중 프로세스 (8)** | **다중 프로세스 (10)** |
| 0.001547 seconds | 0.000588 seconds | 0.000531 seconds |

프로세스 8개일 때와 큰 차이는 보이지 않았지만,

여러 번의 테스트 결과, 다중 프로세스 8개보다 10개가 아주 미세하게 빠른 것을 확인했습니다.

이전 실험과 같이 더 큰 차이를 확인하기 위해

‘1부터 1,000까지’가 아닌, ‘1부터 100,000까지’로 조건을 변경하여 실험해보았습니다.

5회차 (단일 프로세스, 8개 프로세스, 10개 프로세스)

텍스트, 스크린샷, 멀티미디어 소프트웨어, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

|  |  |  |
| --- | --- | --- |
| **단일 프로세스** | **다중 프로세스 (8)** | **다중 프로세스 (10)** |
| 0.164597 seconds | 0.000734 seconds | 0.000724 seconds |

(1~100,000까지) + (코드 100회 반복)의 평균

마찬가지로 8개와 10개의 차이는 크지 않았지만, 단일 프로세스와의 차이는 확연하게 드러났습니다.

프로세스 8개와 10개의 차이가 크지 않은 이유에 대해 Chat GPT에 물어보았습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위 답변을 통해 런타임 시간을 줄이기 위해

**프로세스를 추가할 때 걸리는 시간**과 **연산에서 걸리는 시간** 등을 균등하게 고려하여 프로그래밍 해야 한다는 것을 알게 되었습니다.

1. **추가 실험**

이후, 프로세스가 100개라면, 어떻게 작동하는지 궁금하여 실행해보았습니다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

예상대로, 프로세스 생성과 제거에 필요한 시간과 작업 분배 등의 요소 때문에, 시간이 더 걸리는 것을 확인하였습니다.

위 실험과 마찬가지로, 1부터 100,000까지 실험해보았고, 총 100회 반복 후 평균을 산출했습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| **단일 프로세스** | **다중 프로세스 (8)** | **다중 프로세스 (10)** | **다중 프로세스 (100)** |
| 0.164597 seconds | 0.000734 seconds | 0.000724 seconds | 0.006383 seconds |

런타임 시간 순서는 다음과 같습니다.

다중 프로세스 (10) < 다중 프로세스 (8) < 다중 프로세스 (100) < 단일 프로세스

**(추가)**

clock() 함수와 clock\_gettime() 함수의 차이

|  |  |  |
| --- | --- | --- |
|  | **clock()** | **clock\_gettime()** |
| **정확도** | - CPU 시간 측정 함수  - 초당 클럭 수에 해당하는 해상도를 가짐 | - 나노초 단위의 해상도를 가짐  - 정확한 시간 측정에 적함 |
| **시간 기준** | - 프로그램/프로세스 시작 이후 경과된 CPU 시간 측정  - 프로그램이 실행되는 동안의 CPU 시간만을 측정함 | - 시스템 시간 기준으로 측정  - CLOCK\_MONOTONIC을 사용 시 시스템 종료/재부팅되어도 시간이 초기화 되지 않음  - 따라서 안정적으로 시간 측정 가능 |

위 표를 바탕으로 clock\_gettime() 함수 사용 시 시간 측정을 해보았습니다.

조건

- 1~1000까지 계산

- 100회 반복 후 평균 텍스트, 전자제품, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

|  |  |  |  |
| --- | --- | --- | --- |
| **단일 프로세스** | **다중 프로세스 (8)** | **다중 프로세스 (10)** | **다중 프로세스 (100)** |
| 0.003185 seconds | 0.003955 seconds | 0.004431 seconds | 0.016738 seconds |

clock\_gettime() 함수를 사용했을 경우에는 단일 프로세스가 가장 빠른 시간 측정값을 가졌습니다.

그 이유에 대한 Chat GPT의 답변은 다음과 같습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. **기능 추가 코드**

매번 코드를 수정하지 않아도 되도록 변수들을 console 내에서 변경 가능하도록 변경하였습니다.

<변경 가능한 것>

* 프로세스 개수
* 1부터 n까지 계산할 때, n의 값
* 반복 횟수 (평균값 도출을 위한 반복)
* 시간 측정 함수 ( clock\_gettime() 또는 clock() )

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

1. *#include* <stdio.h>
2. *#include* <stdlib.h>
3. *#include* <sys/types.h>
4. *#include* <unistd.h>
5. *#include* <sys/wait.h>
6. *#include* <time.h>
7. *#define* MULVALUE 7
8. bool isClockGettime;
9. *// process가 i\*MULVALUE 연산을 하기 위한 함수*
10. void process\_task(int *start*, int *end*, int *process\_num*) {
11. *// printf("Process %d - Start: %d, End: %d\n", process\_num, start, end);*
12. *// 연산*
13. *for* (int i = start; i <= end; ++i) {
14. printf("%d \* %d = %d\n", i, MULVALUE, i \* MULVALUE);
15. }
16. *// 아래 printf는 시간 측정 시 주석 처리하였습니다.*
17. printf("Process PID:%d\n", (int)getppid()); *// Process PID Check*
18. printf("Process %d end\n", process\_num);
19. *// printf("---------------------------------------------\n");*
20. }
21. int main() {
22. int process\_count;
23. printf("-----------------------------------------\n");
24. printf("Process Count: "); scanf("%d", &process\_count);
25. printf("-----------------------------------------\n");
26. int nRange;
27. printf("Range (1~n): "); scanf("%d", &nRange);
28. printf("-----------------------------------------\n");
29. int repeat;
30. printf("Repeat: "); scanf("%d", &repeat);
31. printf("-----------------------------------------\n");
32. int tf;
33. printf("isClockGettime?\n");
34. printf("- Use 'clock()': 0\n");
35. printf("- Use 'clock\_gettime()': 1\n");
36. printf("input: "); scanf("%d", &tf);
37. printf("-----------------------------------------\n");
38. isClockGettime = tf;
39. *// 반복 후 평균을 내기 위해 REPEAT회 결과값을 더하고 저장할 변수*
40. double totalSum = 0;
41. *// 반복 횟수*
42. *// REPEAT만큼 반복*
43. *for*(int testCase = 0; testCase < repeat; testCase++) {
44. *// startTime: 시작 시간*
45. *// endTime: 끝 시간*
46. struct timespec startGetTime, endGetTime;
47. clock\_t startTime, endTime;
48. *if*(isClockGettime) {
49. *// struct timespec startTime, endTime;*
50. clock\_gettime(CLOCK\_MONOTONIC, &startGetTime);
51. }
52. *else* {
53. *// clock\_t startTime, endTime;*
54. startTime = clock();
55. }
56. *// totalTime: startTime과 endTime을 연산 후 나온 결과 (실행 시간)*
57. double totalTime;
58. *// pid: 프로세스 ID 저장*
59. pid\_t pid;
60. *// start: 연산할 값의 시작 값*
61. *// end: 연산할 값의 마지막 값 (+ 1)*
62. *// range: 각 프로세스의 처리 범위*
63. int start, end, range, i;
64. range = nRange / process\_count;
65. *// 프로세스의 개수만큼 반복*
66. *for* (i = 0; i < process\_count; ++i) {
67. start = i \* range + 1;
68. end = (i + 1) \* range;
70. *// 자식 프로세스 생성*
71. pid = fork();
72. *if* (pid < 0) { *// fork 실패*
73. perror("fork");
74. exit(1);
75. } *else* *if* (pid == 0) { *// 자식 프로세스*
76. *// process\_task 함수로 이동*
77. *// process\_task(시작 값, 마지막 값, 프로세스 번호)*
78. process\_task(start, end, i);
79. exit(0);
80. }
81. }
82. *// 자식 프로세스 종료 대기*
83. *for* (i = 0; i < process\_count; ++i) {
84. wait(NULL);
85. }
86. *if*(isClockGettime) {
87. clock\_gettime(CLOCK\_MONOTONIC, &endGetTime);
88. totalTime = (endGetTime.tv\_sec - startGetTime.tv\_sec) + (endGetTime.tv\_nsec - startGetTime.tv\_nsec) / 1e9;
89. }
90. *else* {
91. endTime = clock();
92. totalTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;
93. }
94. printf("Total Time: %f seconds\n", totalTime);
95. totalSum += totalTime;
96. }
97. *// 평균 시간 계산 후 출력*
98. printf("=====================================================\n");
99. printf("[Average time]\n");
100. printf("- Process Count: %d\n", process\_count);
101. printf("- Range: 1~%d\n", nRange);
102. printf("- Repeat: %d\n", repeat);
103. printf("- Use Function: %s\n", ((isClockGettime==1) ? "clock\_gettime()" : "clock()"));
104. printf("=====================================================\n");
105. printf("=> %f seconds\n", totalSum / repeat);
106. printf("=====================================================\n");
107. *return* 0;
108. }