گزارش دستورکار پنجم آزمایشگاه سیستم‌های عامل

نگار موقتیان، ۹۸۳۱۰۶۲

۱. انجام محاسبات به صورت سریال

در این قسمت از آزمایش می‌خواهیم برنامه‌ای بنویسیم که به صورت سریال مطابق دستورکار آرایهٔ hist را پر کند، بنابراین در این مرحله برای اجرای تمام محاسبات تنها از یک پردازه استفاده می‌کنیم. ساختار برنامهٔ نوشته شده بدین منظور در ادامه آمده‌است.

در ابتدای برنامه کتابخانه‌های مورد نیاز اضافه شده‌اند.

پس از آن مطابق توضیحات دستورکار یک آرایهٔ ۲۵ تایی به نام hist و یک متغیر به نام samples تعریف شده‌است که تعداد نمونه‌های آزمایش را در خود ذخیره می‌کند (این متغیر با توجه به ورودی کاربر مقداردهی خواهد شد).

در ادامه تابع printHistogram نیز مانند آنچه در دستورکار آمده تعریف شده‌است، با این تفاوت که یک عدد صحیح scale را به عنوان ورودی خود می‌پذیرد. زمانی که تعداد نمونه‌های آزمایش زیاد می‌شود مقدار هر خانه از آرایهٔ hist نیز به طبع افزایش خواهد یافت. به ازای تعداد نمونه‌های بسیار بزرگ مقدار بعضی از خانه‌های این آرایه به قدری زیاد می‌شود که نمایش آن در یک خط ترمینال ممکن نیست. بنابراین یک متغیر scale تعریف شده‌است تا تمام این داده‌ها را به نسبتی کوچک کند. برای انتخاب این ضریب نیز به این صورت عمل شده‌است که اگر تعداد نمونه‌ها کم‌تر از ۵۰۰ بود به آن‌ها ضریب ۱ می‌دهیم و در غیر این صورت آن‌ها را تقسیم بر (۵۰۰ / تعداد نمونه‌ها) می‌کنیم. در این صورت مطمئن هستیم تعداد ستاره‌های چاپ شده از ۵۰۰ عدد بیش‌تر نمی‌شود.

پس از آن در تابع main ابتدا تابع ()srand را صدا می‌زنیم تا یک هسته برای تولیدکنندهٔ اعداد تصادفی تعیین کنیم. در غیر این صورت با هر بار اجرای برنامه اعداد تصادفی تولید شده یکسان خواهند بود.

در ادامه مقدار آرگومان اول داده شده به برنامه را به عنوان تعداد نمونه‌ها تعریف می‌کنیم و در صورتی که این آرگومان به برنامه داده نشده بود پیغام مناسب داده و برنامه را خاتمه می‌دهیم.

سپس در یک حلقهٔ تو در تو به ازای هر نمونه ۱۲ عدد تصادفی تولید کرده و مطابق دستور کار مقدار counter را تغییر می‌دهیم تا نهایتاً این نمونه را در یکی از خانه‌های آرایهٔ hist قرار دهیم.

در نهایت نیز با استفاده از تابع printHistogram نمودار متناظر با مقادیر موجود در آرایهٔ hist را رسم می‌کنیم.

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

int hist[25], samples;

// print the final histogram. the variable "scale" is used as to make sure the
printed stars fit in the terminal window
void printHistogram(int scale) {
	printf("\n====================================");
	for (int i = 0; i < 25; i++) {
		for (int j = 0; j < hist[i] / scale; j++)
			printf("*");
		printf("\n");
	}
	printf("====================================\n");
}

int main(int argc, char *argv[]) {
	srand(time(0));

	if (argc < 2) { // check if all the required parameters are passed
		printf("Please pass the number of samples as argument.\n");
		return -1;
	}
	samples = atoi(argv[1]);

	// record the starting time
	struct timeval begin, end;
	gettimeofday(&begin, 0);

	// fill the hist array
	for (int i = 0; i < samples; i++) {
		int counter = 0;
		for (int j = 0; j < 12; j++)
			counter += (rand() % 101 >= 49) ? 1 : -1;
		hist[counter + 12]++;
	}

	// compute the elapsed time in milliseconds and print it
	gettimeofday(&end, 0);
	long seconds = end.tv_sec - begin.tv_sec;
	long microseconds = end.tv_usec - begin.tv_usec;
	double elapsed = (seconds + microseconds*1e-6) * 1e3;
	printf("\nFinished Processing in %.3f milliseconds.\n", elapsed);

	// plot the final result
	printHistogram((samples >= 500) ? samples / 500 : 1);

	return 0;
}
```

همچنین زمان را در شروع فرآیند برنامه خوانده و در یک struct از جنس timeval ذخیره می‌کنیم. همین کار را در انتهای فرآیند نیز انجام داده و اختلاف این دو زمان را پیدا می‌کنیم تا زمان سپری شده برای اجرای عملیات برنامه را محاسبه کنیم. برای اندازه‌گیری زمان در این قسمت از برنامه از لینک زیر کمک گرفته شده‌است:

https://levelup.gitconnected.com/8-ways-to-measure-execution-time-in-c-c-48634458d0f9

در نهایت خروجی این برنامه به ازای مقادیر خواسته شده در دستورکار به صورت زیر است:

```
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ gcc -o single-process OSLab05_single-process.c
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./single-process
Please pass the number of samples as argument.
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./single-process 5000

Finished Processing in 1.327 milliseconds.

====================================

*

******

**********************

**************************************************

**********************************************************************************

**********************************************************************************************************************

**************************************************************************************

******************************************************************

*******************************

*********

**


==================================
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$
```

در ابتدا برنامه را کامپایل کرده‌ایم.

سپس در اجرای اول به برنامه آرگومانی نداده‌ایم و همانطور که دیده می‌شود پیغام مناسبی چاپ شده‌است.

پس از آن تعداد نمونه‌ها ۵,۰۰۰ عدد در نظر گرفته شده و محاسبات در ۱/۳۲۷ میلی ثانیه انجام شده‌است.

```
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./single-process 50000

Finished Processing in 13.818 milliseconds.

====================================

*

******

**********************

**************************************************

**********************************************************************************

**********************************************************************************************************************

**********************************************************************************************************

**************************************************************

******************************

*********

*


====================================
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$
```

این بار تعداد نمونه‌ها ۵۰,۰۰۰ عدد در نظر گرفته شده و محاسبات در ۱۳/۸۱۸ میلی ثانیه انجام شده‌است.

```
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./single-process 500000

Finished Processing in 123.588 milliseconds.

====================================

*

******

**********************

**************************************************

**********************************************************************************

**********************************************************************************************************************

**********************************************************************************************

**************************************************************

******************************

**********

*


====================================
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$
```

و در نهایت تعداد نمونه‌ها ۵۰۰,۰۰۰ عدد در نظر گرفته شده و برنامه در ۱۲۳/۵۸۸ میلی ثانیه انجام شده‌است.

بنابراین جدول خواسته شده را می‌توان با مقادیر زیر پر کرد:

تعداد نمونه	۵,۰۰۰	۵۰,۰۰۰	۵۰۰,۰۰۰
زمان اجرا (میلی ثانیه)	۱/۳۲۷	۱۳/۸۱۸	۱۲۳/۵۸۸

۲. انجام محاسبات به صورت موازی

حال در این قسمت از آزمایش می‌خواهیم برنامه را به صورتی بنویسیم که در آن چندین پردازه به صورت موازی بر روی آرایهٔ hist کار کنند. در این صورت با تقسیم کار میان پردازه‌ها می‌تواند سرعت اجرای برنامه را افزایش داد. بدین منظور این بار آرایهٔ hist را به صورت shared memory تعریف می‌کنیم تا تمامی پردازه‌ها به آن دسترسی داشته باشند و بتوانند به طور همزمان بر روی آن داده بنویسند و در نهایت پردازهٔ پدر نیز بتواند مقدار آن را بخواند تا پاسخ نهایی را چاپ کند. ساختار برنامهٔ نوشته شده بدین منظور در ادامه آمده‌است.

در ابتدای برنامه کتابخانه‌های مورد نیاز اضافه شده‌اند.

پس از آن متغیرهای samples، تعداد نمونه‌ها، processes، تعداد پردازه‌ها، pid، pid ای که تابع fork در ادامه برمی‌گرداند (دلیل مقداردهی اولیهٔ آن به ۱ توضیح داده خواهد شد) و memoryID که مسئول ذخیره‌سازی id مربوط به shared memory ساخته شده است تعریف شده‌اند.

در ادامه تابع printHistogram نیز درست مانند قسمت قبل تعریف شده‌است، با این تفاوت که این بار آرایهٔ hist به آن پاس داده شده‌است، زیرا دیگر یک آرایهٔ global نیست و آن را در طول اجرای برنامه از طریق shared memory دریافت می‌کنیم.

پس از آن در تابع main باز هم ابتدا تابع ()srand را صدا می‌زنیم تا نتایج تولید شده کاملاً تصادفی باشند.

سپس در ادامه مقدار آرگومان اول و دوم داده شده به برنامه را به عنوان تعداد نمونه‌ها و تعداد پردازه‌ها تعریف می‌کنیم و در صورتی که این آرگومان‌ها به برنامه داده نشده بودند پیغام مناسب داده و برنامه را خاتمه می‌دهیم.

حال shared memory ای که قرار است پردازه‌ها به طور مشترک بر روی آن کار کنند را تعریف می‌کنیم. برای این کار از متن آزمایش قبلی دستورکار و همچنین سایت زیر برای تکمیل ورودی‌های تابع shmget استفاده شده‌است:

https://www.mkssoftware.com/docs/man3/shmget.3.asp

سپس به تعداد پردازه‌ها (که آن را به عنوان ورودی دریافت کرده‌ایم) تابع fork را بر روی پردازهٔ اصلی صدا می‌زنیم تا فرزندان آن به تعداد مناسب ایجاد شوند. برای تشخیص این که پردازه‌ای که در آن هستیم پردازهٔ اصلی برنامه و والد باقی پردازه‌هاست از متغیر pid استفاده می‌شود. این متغیر ابتدا با ۱ (که عددی بزرگ‌تر از صفر است) مقداردهی شده‌است تا بار اول وارد if شده و fork را اجرا کنیم. از آن به بعد با هر بار اجرای fork متغیر pid را آپدیت می‌کنیم، از این طریق در پردازه‌های فرزند همواره pid برابر با صفر و در پردازهٔ اصلی این عدد هر بار یک عدد بزرگ‌تر از صفر خواهد بود (به شرط اجرای موفقیت‌آمیز fork).

حال که تمام پردازه‌هایی که می‌خواستیم ایجاد شدند، برای هر یک از آن‌ها آرایهٔ hist را از shared memory از طریق ID آن بازیابی کرده و ذخیره می‌کنیم.

پس از آن بررسی می‌کنیم که داخل پردازهٔ والد هستیم یا یکی از پردازه‌های فرزند.

۱. اگر داخل پردازهٔ والد بودیم ابتدا زمان فعلی را به عنوان شروع زمان اجرا رکورد می‌کنیم (پیش از این در حال ایجاد پردازه‌ها بودیم و در این برنامه تنها زمان نیاز برای انجام محاسبات اندازه‌گیری شده‌است). سپس به ازای تمام فرزندان wait می‌کنیم تا تمام محاسبات مربوطه پایان پذیرند. پس از آن مانند بخش قبلی آزمایش زمان سپری شده در این مدت را اندازه‌گیری کرده و چاپ می‌کنیم. در نهایت نیز آرایهٔ پر شده توسط پردازه‌های فرزند را توسط تابع printHistogram چاپ کرده و shareed memory ای که در حافظهٔ اصلی رزور کرده بودیم را آزاد می‌کنیم.
۲. در غیر این صورت اگر داخل پردازهٔ فرزند بودیم سهم خود از محاسبات را انجام می‌دهیم (در این حالت تعداد نمونه‌هایی که باید ایجاد شوند تعداد نمونه‌ها تقسیم بر تعداد پردازه‌ها خواهد بود).

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


int samples, processes, memoryID, pid = 1;

// print the final histogram. the variable "scale" is used as to make sure the
printed stars fit in the terminal window
void printHistogram(int *hist, int scale) {
      printf("\n====================================");
```

```
	for (int i = 0; i < 25; i++) {
		for (int j = 0; j < hist[i] / scale; j++)
			printf("*");
		printf("\n");
	}
	printf("====================================\n");
}

int main(int argc, char *argv[]) {
	srand(time(0));

	if (argc < 3) { // check if all the required parameters are passed
		printf("Please pass the number of samples and processes as argument.\n");
		return -1;
	}
	samples = atoi(argv[1]);
	processes = atoi(argv[2]);

	// initiate the shared memory
	memoryID = shmget(IPC_PRIVATE, 25 * sizeof(int), IPC_CREAT | 0666);

	// create child processes
	for (int i = 0; i < processes; i++)
		if (pid > 0) // fork only if we are in the main process
			pid = fork();
		else
			break;

	// retrieve the shared memory
	int *hist = (int *) shmat(memoryID, NULL, 0);

	if (pid > 0) { // we are in the main process
		// record the starting time
		struct timeval begin, end;
		gettimeofday(&begin, 0);

		// wait for all children to finish their process
		for (int i = 0; i < processes; i++)
			wait(NULL);

		// compute the elapsed time in milliseconds and print it
		gettimeofday(&end, 0);
		long seconds = end.tv_sec - begin.tv_sec;
		long microseconds = end.tv_usec - begin.tv_usec;
		double elapsed = (seconds + microseconds*1e-6) * 1e3;
		printf("\nFinished Processing in %.3f milliseconds.\n", elapsed);

		// plot the final result
		printHistogram(hist, (samples >= 500) ? samples / 500 : 1);
```

```
		// free the shared memory
		shmctl(memoryID, IPC_RMID, NULL);
	}
	else { // we are in one of the child processes, so we need to fill the hist
array
		for (int i = 0; i < samples / processes; i++) {
			int counter = 0;
			for (int j = 0; j < 12; j++)
				counter += (rand() % 101 >= 49) ? 1 : -1;
			*(hist + counter + 12) = *(hist + counter + 12) + 1;
		}
	}

	return 0;
}
```

در نهایت خروجی این برنامه به ازای مقادیر خواسته شده در دستورکار به صورت زیر است:

```
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ gcc -o multi-process OSLab05_multi-process.c
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./multi-process
Please pass the number of samples and processes as argument.
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./multi-process 5000 5

Finished Processing in 1.150 milliseconds.

====================================

*

*****

******************

**************************************************

**********************************************************************************

**********************************************************************************************************************

*********************************************************************************

************************************************************************

********************************

**********




====================================
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$
```

در ابتدا برنامه را کامپایل کرده‌ایم.

سپس در اجرای اول به برنامه آرگومانی نداده‌ایم و همانطور که دیده می‌شود پیغام مناسبی چاپ شده‌است.

پس از آن تعداد نمونه‌ها ۵,۰۰۰ عدد و تعداد پردازه‌ها ۵ عدد در نظر گرفته شده و محاسبات در ۱/۱۵۰ میلی ثانیه انجام شده‌است.

```
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./multi-process 50000 5

Finished Processing in 10.891 milliseconds.

====================================


******

**********************

************************************************

********************************************************************************

****************************************************************************************************************

********************************************************************************************************

************************************************************

****************************

**********

*


====================================
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$
```

این بار تعداد نمونه‌ها ۵۰,۰۰۰ عدد و تعداد پردازه‌ها ۵ عدد در نظر گرفته شده و محاسبات در ۱۰/۸۹۱ میلی ثانیه انجام شده‌است.

```
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$ ./multi-process 500000 5

Finished Processing in 82.351 milliseconds.

====================================

*

******

**********************

************************************************

********************************************************************************

****************************************************************************************************************

********************************************************************************

************************************************************

****************************

**********

*


====================================
maxeu@ubuntu:~/Desktop/OS Lab/Lab05$
```

و در نهایت تعداد نمونه‌ها ۵۰۰,۰۰۰ عدد و تعداد پردازه‌ها ۵ عدد در نظر گرفته شده و برنامه در ۸۲/۳۵۱ میلی ثانیه انجام شده‌است.

بنابراین جدول خواسته شده را می‌توان با مقادیر زیر پر کرد:

تعداد نمونه	۵,۰۰۰	۵۰,۰۰۰	۵۰۰,۰۰۰
زمان اجرا (میلی ثانیه)	۱/۱۵۰	۱۰/۸۹۱	۸۲/۳۵۱

۳. آیا این برنامه درگیر شرایط مسابقه می‌شود؟ چگونه؟ اگر جوابتان مثبت بود راه حلی برای آن بیابید.

بله؛ از آن‌جایی که تمام پردازه‌های فرزند به طور همروند بر روی آرایهٔ hist مقدار می‌نویسند امکان به وجود آمدن race condition وجود دارد. به طور مشخص در یکی از خطوط برنامه داریم:

```
*(hist + counter + 12) = *(hist + counter + 12) + 1;
```

حال اگر مقدار counter در دو پردازه در یک زمان یکسان باشد، یکی مقدار این خانه از hist را خوانده، آن را تغییر دهد، سپس پیش از این که مقدار جدید این خانه را در آن بریزد پردازه‌ای دیگر نیز همین خانه از آرایه را بخواند تا آن را تغییر دهد، نتیجهٔ مورد انتظار را نخواهیم داشت.

البته از آن‌جایی که حجم نمونه‌ها در این آزمایش زیاد بوده و احتمال این که برای دو پردازه به طور همزمان متغیر counter یک مقدار یکسان پیدا کند کم است، خلل جدی‌ای در روند آزمایش ایجاد نخواهد شد. اما برای رفع این مشکل می‌توان از یکی از مکانیزم‌های همگام سازی استفاده کرد.

برای مثال یکی از معروف‌ترین مکانیزم‌هایی که برای همگام‌سازی پردازه‌ها استفاده می‌شود استفاده از semaphore است. برای استفاده از آن ابتدا یک سمافور با استفاده از shared memory ID ای که داریم ایجاد می‌کنیم. سپس هر زمان که می‌خواستیم داده‌ای بر روی shared memory بنویسیم ابتدا از تابع sem_wait() استفاده می‌کنیم. این تابع بررسی می‌کند که سمافور قفل شده‌است یا خیر (قفل بودن یا نبودن آن با مقدار درون آن مشخص می‌شود. برای مثال اگر مقدار آن 1 بود آزاد و اگر 1- بود قفل است). اگر قفل نبود آن را قفل کرده و بر روی حافظه مقدار دلخواه را می‌نویسد، در غیر این صورت تا زمانی که قفل آن توسط پردازه‌ای که آن را قفل کرده و با استفاده از دستور sem_post() آزاد شود، بلاک می‌شود.

لیست کامل‌تری از این مکانیزم‌ها در لینک زیر آمده‌است:

https://stackoverflow.com/questions/53736985/how-synchronization-is-done-in-shared-memory-data-linux-c

۴. مقایسهٔ روش اول و دوم و بررسی میزان افزایش سرعت:

با توجه به جداول پر شده در قسمت اول و دوم می‌توان نتایج زیر را بدست آورد:

تعداد نمونه	۵,۰۰۰	۵۰,۰۰۰	۵۰۰,۰۰۰
افزایش سرعت	۱۵/۳۹٪	۲۶/۸۷٪	۵۰/۰۷٪

همانطور که دیده می‌شود افزایش سرعت قابل توجهی داشته‌ایم. به علاوه به نظر می‌رسد هر چه تعداد نمونه‌ها بیش‌تر شده، استفاده از چند پردازه به جای یک پردازه به صرفه‌تر است. زیرا از طرفی استفاده از پردازه‌ها به دلیل اجرای موازی کارها سرعت برنامه را افزایش می‌دهد و از طرف دیگر کار کردن با چندین پردازه به جای یک پردازه برای سیستم سرباری ایجاد می‌کند. این سربار با افزایش تعداد نمونه‌ها (به شرط ثابت بودن تعداد پردازه‌ها) ثابت می‌ماند، در حالی که تسریع به دلیل اجرای موازی محاسبات خود را بیش‌تر نشان می‌دهد. لذا مشاهده می‌کنیم که هر چه تعداد نمونه‌ها افزایش یافته‌اند، میزان افزایش سرعت نیز افزایش یافته‌است.