ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ

Սվետա Ավետիսյան, Անի Քոչարյան
Մինաս Հովհաննիսյան

«WINDOWS» ՕՊԵՐԱՑԻՈՆ ՀԱՄԱԿԱՐԳԵՐ

(ուսումնամեթոդական ձեռնարկ)

ԵՐԵՎԱՆ
ԵՊՀ ՀՐԱՏԱՐԱԿՉՈՒԹՅՈՒՆ
2021

ՀՏԴ 004.451(07)
ԳՄԴ 32.973.26-018.2g7
Ա 791

Հրատարակության է երաշխավորել
ԵՊՀ ինֆորմատիկայի և կիրառական մաթեմատիկայի
ֆակուլտետի գիտական խորհուրդը:

Ավետիսյան Սվետա, Քոչարյան Անի,
Հովհաննիսյան Մինաս
Ա 791 «Windows» օպերացիոն համակարգեր *(ուսումնամեթոդական ձեռնարկ)*/ Ս. Ավետիսյան, Ա. Քոչարյան, Մ. Հովհաննիսյան: -Եր., ԵՊՀ հրատ., 2021, 90 էջ:

Ձեռնարկում ներկայացվում է «Microsoft Windows» օպերացիոն համակարգերի ընտանիքի և դրանց կողմից օգտագործվող կիրառությունների ծրագրավորման ինտերֆեյսը (Application Programming Interface, API) : Նկարագրվում են «Windows» օպերացիոն համակարգերի հիմնական հասկացությունները՝ ֆայլային համակարգ, պրոցեսներ, հոսքեր, սինքրոնացման մեթոդներ, միջուկի օբյեկտներ, նկարագրիչներ և այլն, ու դրանց օգտագործման համար նախատեսված համակարգային կանչերը: Բերված են այդ հասկացությունների և համապատասխան համակարգային կանչերի կիրառության, ինչպես նաև տիպային խնդիրների լուծման օրինակներ:

Նախատեսված է ինֆորմատիկայի և կիրառական մաթեմատիկայի ֆակուլտետի երկրորդ և երրորդ կուրսերում անցկացվող «Օպերացիոն համակարգեր» դասընթացի գործնական պարապմունքների ընթացքում կիրառման համար: Ձեռնարկում բերված օրինակները կարդալու և ընկալելու համար անհրաժեշտ է ծանոթ լինել «C/C++» լեզվի հիմունքներին, ինչպես նաև «Visual Studio» ծրագրային միջավայրին:

ՀՏԴ 004.451(07)
ԳՄԴ 32.973.26-018.2g7

ISBN 978-5-8084-2515-6

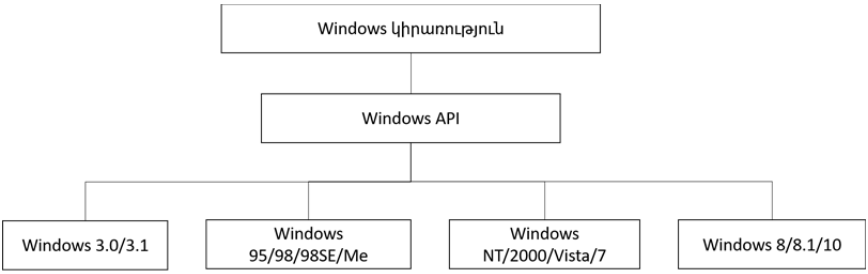
Բովանդակություն

ՆԵՐԱԾՈՒԹՅՈՒՆ

«Windows»-ը ժամանակակից օպերացիոն համակարգ է, որն աշխատում է ինչպես անհատական համակարգիչներում, նոթբուքերում, պլանշետներում և հեռախոսներում, այնպես էլ սերվերային հաշվիչ մեքենաներում: Ձեռնարկի պատրաստման պահին «Windows»-ի վերջին տարբերակը «Windows 10»-ն է:

Windows օպերացիոն համակարգերի նախահայրը MS-DOS (Microsoft Disk Operating System) համակարգն է, որի առաջին տարբերակն ի հայտ է եկել դեռևս 1981 թ.: «Microsoft» ընկերության կողմից «Windows» օպերացիոն համակարգի մշակման գործընթացը կարելի է բաժանել 4 հիմնական շրջանների՝ «MS-DOS», «MS-DOS»-ի հիմքով «Windows», «NT» (New Technology) հիմքով «Windows» և ժամանակակից «Windows»:

Բոլոր «Windows» օպերացիոն համակարգերի համար հասանելի է «Microsoft»-ի կիրառությունների ծրագրավորման ինտերֆեյսը՝ «Windows API» (Application Programming Interface): Վերջինիս օգտագործումը թույլ է տալիս մշակել կիրառություններ, որոնք աշխատում են «Windows»-ի համարյա բոլոր տարբերակների հետ (Նկ. 1):



Նկար 1: «Windows API»-ի կապը «Windows» օպերացիոն համակարգի տարբերակների հետ:

ԳԼՈՒԽ 1

«WINDOWS» ԿԻՐԱՌՈՒԹՅԱՆ ՄՇԱԿՈՒՄԸ

Այս գլխում մատուցվում են «Windows» կիրառության մշակման համար անհրաժեշտ տարրական տեղեկություններ: Նախ տրվում են ներածական գիտելիքներ՝ «Unicode» ստանդարտի և «Windows API»-ի վրա դրա ազդեցության մասին: Այնուհետև կատարվում է նախնական ծանոթացում սխալների մշակման մեխանիզմին, իսկ հետո՝ հրամայական տողի, միջավայրի փոփոխականների և ընթացիկ կատալոգի հետ աշխատանքներին:

1.1. «Unicode»

Տարիներ շարունակ սիմվոլային տողերը կոդավորվել են որպես մեկբայթանոց սիմվոլների հաջորդականություն, որն ավարտվում է 0-ական սիմվոլով: Մշակվել են բազմաթիվ գրադարանային ֆունկցիաներ, որոնք աշխատում են այդ ձևով կոդավորված սիմվոլային տողերի հետ, օրինակ՝ «C/C++» string.h-ի հայտնի strlen, strcat, strcpy և այլն: Սակայն գոյություն ունեն այնքան շատ սիմվոլներ պարունակող լեզուներ և գրառման եղանակներ (օրինակ՝ ճապոնական հիերոգլիֆները), երբ մեկ բայթը (որի միջոցով կարելի է կոդավորել առավելագույնը 256 սիմվոլ) այլևս չի բավարարում: Այդպիսի լեզուների սիմվոլները կոդավորելու նպատակով ստեղծվեցին սկզբում «DBCS» (double byte character sets), իսկ ավելի ուշ՝ «Unicode» կոդավորման եղանակները:

«Unicode»-ը 1988 թ. «Apple» և «Xerox» ընկերությունների կողմից մշակված ստանդարտ է: Բոլոր սիմվոլները ներկայացված են երկուբայթանոց արժեքներով, ինչը թույլ է տալիս կոդավորել 65 536 սիմվոլ 256-ի փոխարեն: Այդ սիմվոլների բազմությունը տրոհված է մի քանի խմբերի, օրինակ՝ 0000-007F կոդային դիրքերը ASCII սիմվոլներն են, 0400-04FF-ը՝ կիրիլիցայի սիմվոլները, իսկ 0530-058F-ը՝

հայերեն սիմվոլները: 29 000 կոդային դիրքեր պահուստավորված են հետագա օգտագործման համար:

2000 թվականից սկսած՝ «Windows» օպերացիոն համակարգն ամբողջովին հիմնված է «Unicode» կոդավորման ստանդարտի վրա: Բոլոր հիմնական ֆունկցիաները, որոնք ստեղծում են պատուհան, դուրս են բերում տեքստ, աշխատում են սիմվոլային տողերի հետ, սպասում են, որ իրենց «Unicode» սիմվոլային տողեր փոխանցվեն: Եթե որևէ «Windows» ֆունկցիայի «ANSI» սիմվոլային տող է փոխանցվում, ապա վերջինս նախ ձևափոխվում է «Unicode»-ի, հետո նոր փոխանցվում է օպերացիոն համակարգին: Սա տեղի է ունենում օգտագործողի համար աննկատ ձևով, բայց, իհարկե, հիշողության և ժամանակի լրացուցիչ ծախս է առաջացնում:

String.h գրադարանը փոփոխվել է, որպեսզի հնարավոր լինի «C/C++» գրադարանային ֆունկցիաներն օգտագործել «Unicode» սիմվոլային տողերի համար. գրադարանում ավելացվել են wchar_t տիպը և այդ տիպի սիմվոլային տողերի հետ աշխատող ֆունկցիաներ՝ wcslen, wcscat, wcscpy և այլն (wcs-ը՝ wide character set):

«Windows» կիրառություն մշակելիս ծրագրավորողը հնարավորություն ունի ստեղծելու մեկ կոդի ֆայլ, որը կկոմպիլացվի ինչպես «Unicode»-ի օգտագործմամբ, այնպես էլ առանց դրա: «Unicode» օգտագործելու համար անհրաժեշտ է սահմանել «UNICODE» մակրոսը բոլոր #include-ներից առաջ կամ կոմպիլացման ժամանակ: Նշենք, որ «Visual Studio 2015»-ից սկսած՝ այդ մակրոսը լռությամբ սահմանված է ստեղծվող նախագծերում:

Ակնհայտ է, որ str* կամ wcs* ֆունկցիաները բացահայտորեն կանչող կոդը հնարավոր չէ կոմպիլացնել և՛ «ANSI», և՛ «Unicode»-ի օգտագործմամբ: Այդ պատճառով խորհուրդ է տրվում կիրառելու tchar.h ֆայլը: Այն թույլ է տալիս ունիվերսալ կոդ գրել, որը կկոմպիլացվի ինչպես «ANSI», այնպես էլ «Unicode» օգտագործելիս: tchar.h ֆայլը բաղկացած է str* և wcs* ֆունկցիաների կանչերը փոխարինող մի շարք մակրոսներից: Օրինակ՝ այդ ֆայլում սահմանված է _tcscat մակրոսը, որը հղվում է strcat ֆունկցիային, եթե կոդում սահմանված

չէ «UNICODE»-ը, և wcscat ֆունկցիային, եթե այն սահմանված է: tchar.h ֆայլում սիմվոլային տողերի հետ աշխատող բոլոր ստանդարտ ֆունկցիաներն ունեն իրենց համարժեք տարբերակները (Աղ. 1): Փաստորեն, այս մակրոսների օգտագործումը թույլ է տալիս գրել կոդ, որն առանց որևէ փոփոխության կկոմպիլացվի ինչպես «ANSI»-ի, այնպես էլ «Unicode»-ի համար:

char տիպ <string.h> ֆայլ	**wchar_t տիպ <string.h> կամ <wchar.h> ֆայլ**	**<tchar.h> ֆայլ**
strlen	wcslen	_tcslen
strcat	wcscat	_tcscat
strcpy	wcscpy	_tcscpy
strcmp	wcscmp	_tcscmp
strstr	wcsstr	_tcsstr

Աղյուսակ 1: C/C++-ական սիմվոլային տողերի հետ աշխատող գրադարանային որոշ ֆունկցիաներ և դրանց համարժեք մակրոսներ <tchar.h> ֆայլում:

Սակայն սիմվոլների կոդավորման տեսանկյունից այսքանը դեռ բավարար չէ ունիվերսալ կոդ գրելու համար: «Windows API»-ի համար սահմանված են CHAR-ը՝ C/C++-ական char-ի, և WCHAR-ը՝ C/C++-ական wchar_t-ի համար: tchar.h-ում որպես ընդհանրացված սիմվոլային տիպ սահմանված է TCHAR մակրոսը, որն էլ պետք է օգտագործել որպես սիմվոլային տիպ: Սահմանված են նաև _T կամ _TEXT մակրոսները՝ ընդհանրացված տողային հաստատուններ ունենալու համար: Օրինակ՝ TCHAR* str = _T("string"); սիմվոլային տողի սահմանումը կկոմպիլացվի ինչպես «UNICODE»-ի սահմանված լինելու, այնպես էլ չլինելու դեպքում:

Համանման տրամաբանությամբ ներմուծվել է նաև wmain մուտքային ֆունկցիան՝ «Unicode» կոդավորմամբ արգումենտներ ստանալու համար: tchar.h-ում սահմանված է նաև _tmain մակրոսը, որն էլ այսուհետ կօգտագործվի ձեռնարկի օրինակներում:

1.2. Սխալների մշակումը

Նախքան «Microsoft Windows»-ի տրամադրած ֆունկցիաներն ուսումնասիրելը դիտարկենք, թե ինչպես է դրանցում իրականացվում սխալների մշակումը:

«Windows API»-ի ֆունկցիաները կանչվելիս նախ կատարում են փոխանցված պարամետրերի ստուգում, հետո միայն սկսում իրենց աշխատանքը: Եթե փոխանցված պարամետրերն անթույլատրելի են, կամ որևէ պատճառով տվյալ գործողությունը հնարավոր չէ կատարել, ֆունկցիան սխալի մասին վկայող արժեք է վերադարձնում: Աղյուսակ 2-ում բերված են «Windows API»-ի ֆունկցիաների մեծամասնության կողմից վերադարձվող արժեքների տիպերը:

Տիպ	Նշանակություն
VOID	Ֆունկցիան չի սխալվում:
BOOL	Ֆունկցիան վերադարձնում է TRUE հաջող կատարման, FALSE՝ հակառակ դեպքում:
PVOID/LPVOID	Ֆունկցիան վերադարձնում է հիշողության բլոկի հասցե հաջող կատարման և NULL՝ հակառակ դեպքում:
DWORD/LONG	Ֆունկցիան վերադարձնում է հատուկ իմաստ ունեցող որևէ թիվ[1] հաջող կատարման և 0 կամ -1՝ հակառակ դեպքում:
HANDLE	Ֆունկցիան վերադարձնում է վավեր նկարագրիչ հաջող կատարման և NULL կամ INVALID_HANDLE_VALUE՝ հակառակ դեպքում:

Աղյուսակ 2: «Windows API»-ի ֆունկցիաների մեծամասնության կողմից վերադարձվող արժեքների տիպերը և դրանց նշանակությունը:

[1] «Windows API»-ի ֆունկցիաների և տիպերի մասին լրացուցիչ տեղեկություններ կարելի է ստանալ Microsoft-ի պաշտոնական փաստաթղթերի էջում, docs.microsoft.com:

«Windows»-ի ֆունկցիաները, հայտնաբերելով սխալը, հոսքի լոկալ հիշողության մեջ (տե՛ս 3.3)՝ հատուկ տեղում, գրում է սխալի կոդը: Ամեն սխալին կցված է 32-բիթանոց կոդ[2]: Պարզելու համար, թե կոնկրետ ինչ սխալ է տեղի ունեցել ֆունկցիայի կատարման ընթացքում, օգտագործվում է GetLastError ֆունկցիան, որը վերադարձնում է այդ 32-բիթանոց թիվը: Վերջին սխալի կոդը փոփոխելու համար օգտագործվում է SetLastError ֆունկցիան (Աղ. 3):

Ֆունկցիա	**Նկարագիր**
DWORD GetLastError();	Վերադարձնում է վերջին սխալի կոդը:
VOID SetLastError(**DWORD dwErrCode** **);**	Փոխում է վերջին սխալի կոդը dwErrCode-ով:

Աղյուսակ 3: GetLastError և SetLastError ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Հաճախ կարիք է լինում ստանալ ոչ թե սխալի կոդը, այլ այն նկարագրող տեքստային բացատրությունը: Այդ նպատակով կարելի է օգտագործել FormatMessage ֆունկցիան: Ծրագիր 1-ում[3] ցույց է տրված GetLastError, SetLastError և FormatMessage ֆունկցիաների օգտագործումը՝ վերջին սխալը նկարագրող տեքստի դուրսբերման և վերջին սխալի կոդի փոփոխման համար: Հաջորդ բոլոր ծրագրերում սխալների մշակումը կազմակերպելիս օգտագործվում է error_text_output ֆունկցիան:

[2] «Microsoft»-ի կողմից սահմանված սխալների նույնարկիչները, կոդերը և տեքստային բացատրությունները գտնվում են WinError.h ֆայլում:

[3] Այս և հաջորդ օրինակներում main կամ wmain ֆունկցիաների փոխարեն օգտագործվում է _tmain ընդհանրացված ձևը (տե՛ս 1.1):

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output() {
	LPVOID lpMsgBuf;
	DWORD dw = GetLastError();

	FormatMessage(
		FORMAT_MESSAGE_ALLOCATE_BUFFER |
		FORMAT_MESSAGE_FROM_SYSTEM |
		FORMAT_MESSAGE_IGNORE_INSERTS,
		NULL,
		dw,
		MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
		(LPTSTR)&lpMsgBuf,
		0, NULL);

	_tprintf(_T("Error is: %d - %s\n"), dw, (LPTSTR)lpMsgBuf);

	LocalFree(lpMsgBuf);
}

int _tmain() {
	error_text_output();
	SetLastError(5);
	error_text_output();

	return 0;
}
```

Ծրագիր 1: GetLastError, SetLastError և FormatMessage ֆունկցիաների օգտագործումը:

1.3. Հրամայական տողը, միջավայրի փոփոխականները և ընթացիկ կատալոգը

Կամայական «Windows» կիրառության հետ կապված են հետևյալ 3 գաղափարները՝ հրամայական տող, միջավայրի փոփոխականներ և ընթացիկ կատալոգ։

Հրամայական տողը որպես main (wmain, _tmain) ֆունկցիայի արգումենտներ կիրառությունն ստանում է argc թվային արժեքը և argv սիմվոլային տողերի զանգվածը։ Ընդ որում, argv-ի չափը հենց argc-ն է (Ծրագիր 2)։ Պետք է հաշվի առնել, որ argc-ն միշտ առնվազն 1 է, իսկ argv[0]-ն կատարվող ֆայլի ամբողջական անունն է։

«Windows» կիրառության հետ կապված է նաև միջավայրի փոփոխականների բլոկը։ Վերջինիս վրա ցուցիչ է ստանում main-ը երրորդ արգումենտում՝ envp (Ծրագիր 2)։ Այն սիմվոլային տողերի զանգված է։ Յուրաքանչյուր սիմվոլային տող բաղկացած է միջավայրի փոփոխականի «անուն և արժեք» զույգից՝ հետևյալ ձևաչափով.

```
name1=value1\0
name2=value2\0
…
nameN=valueN\0\0
```

Որևէ միջավայրի փոփոխականի արժեքը կարելի է ստանալ GetEnvironmentVariable ֆունկցիայի, իսկ փոփոխել՝ SetEnvironmentVariable-ի միջոցով (Աղ. 4)։

Ֆունկցիա	Նկարագիր
DWORD GetEnvironmentVariable(**LPTCSTR lpName,** **LPTSTR lpBuffer,** **DWORD nSize** **);**	nSize չափ ունեցող lpBuffer սիմվոլների զանգվածում գրում է lpName անունով միջավայրի փոփոխականի արժեքը։ Վերադարձնում է lpBuffer-ին անհրաժեշտ չափը։

BOOL SetEnvironmentVariable (**LPTCSTR lpName,** **LPTCSTR lpValue** **);**	Փոխում է lpName անունով միջավայրի փոփոխականի արժեքը lpValue-ով: Եթե lpName անունով միջավայրի փոփոխականը չկա, ապա այն ստեղծվում է:

Աղյուսակ 4: GetEnvironmentVariable և SetEnvironmentVariable ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

int _tmain(int argc, TCHAR* argv[], TCHAR** envp) {
	// Command line arguments' output
	for (int i = 0; i < argc; ++i) {
		_tprintf(_T("Command line argument %d is: %s\n"), i,
argv[i]);
	}

	// Environment variable values' output
	int i{};
	while (envp[i] != 0) {
		_tprintf(_T("Environment variable %d is: %s\n"), i, envp[i]);
		++i;
	}

	return 0;
}
```

Ծրագիր 2: Հրամայական տողի արգումենտների և միջավայրի փոփոխականների արժեքների դուրսբերումը:

Ծրագիր 3-ում ցուցադրված է GetEnvironmentVariable և SetEnvironmentVariable ֆունկցիաների օգտագործումը՝ ինչպես ար-

դեն գոյություն ունեցող միջավայրի փոփոխականի արժեքն ստանալու, այնպես էլ նորն ստեղծելու համար։

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

int _tmain() {
	// Getting "path" environment variable value
	const int size = 100;
	TCHAR buffer[size];
	int r = GetEnvironmentVariable(_T("path"), buffer, size);
	if (r == 0) {
		error_text_output();
	}
	else if (r <= size) {
		_tprintf(_T("'path' value is: %s\n"), buffer);
	}
	else {
		_tprintf(_T("Given buffer's size should be: %d\n"), r);
	}

	// Creating new environment variable
	if (!SetEnvironmentVariable(_T("NewVariable"), _T("value1"))) {
		error_text_output();
		return 1;
	}
	r = GetEnvironmentVariable(_T("NewVariable"), buffer, size);
	if (r == 0 || r > size) {
		error_text_output();
		return 1;
	}
	else {
		_tprintf(_T("'NewVariable' value is: %s\n"), buffer);
	}
```

```
	// Changing environment variable value
	if (!SetEnvironmentVariable(_T("NewVariable"), _T("value2"))) {
		error_text_output();
		return 1;
	}
	r = GetEnvironmentVariable(_T("NewVariable"), buffer, size);
	if (r == 0 || r > size) {
		error_text_output();
		return 1;
	}
	else {
		_tprintf(_T("'NewVariable' value is: %s\n"), buffer);
	}

	return 0;
}
```

Ծրագիր 3: GetEnvironmentVariable և SetEnvironmentVariable ֆունկցիաների օգտագործումը:

«Windows» կիրառությունը կատարվում է, այսպես կոչված, ընթացիկ կատալոգում: Ընթացիկ կատալոգում կիրառությունը լռությամբ փնտրում է այն ֆայլերն ու ենթակատալոգները, որոնց ամբողջական անունների փոխարեն ծրագրում օգտագործված են հարաբերական անունները: Պետք է հաշվի առնել, որ ընթացիկ կատալոգը հաճախ չի համընկնում կատարվող ֆայլի տեղակայման հետ: Ընթացիկ կատալոգը կարելի է ստանալ GetCurrentDirectory, իսկ փոփոխել՝ SetCurrentDirectory ֆունկցիաների միջոցով (Աղ. 5):

Ֆունկցիա	**Նկարագիր**
DWORD GetCurrentDirectory(**DWORD nBufferLength,** **LPTSTR lpBuffer** **);**	nBufferLength չափ ունեցող lpBuffer սիմվոլների զանգվածում գրում է ընթացիկ կատալոգի անունը: Վերադարձնում է lpBuffer-ին անհրաժեշտ չափը:

BOOL SetCurrentDirectory (**LPCTSTR lpPathName** **);**	Փոխում է ընթացիկ կատալոգը lpPathName-ով: Եթե այդ անունով կատալոգ չկա, կամ այն չի կարող դառնալ ընթացիկ, ապա ֆունկցիան սխալվում է:

Աղյուսակ 5: GetCurrentDirectory և SetCurrentDirectory ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Խնդիրներ

1. Դուրս բերել հրամայական տողի արգումենտներից յուրաքանչյուրում եղած սիմվոլների քանակները:

2. Դուրս բերել հրամայական տողի արգումենտներից յուրաքանչյուրում հանդիպած 'a' սիմվոլի քանակները:

3. Ինքնուրույն ուսումնասիրել GetCommandLine և CommandLineToArgW ֆունկցիաները և օգտագործել՝ հրամայական տողի արգումենտների արժեքներն ստանալու և դուրս բերելու համար:

4. Հաշվել հրամայական տողի արգումենտների գումարը՝ ենթադրելով, որ փոխանցվել են թվային արժեքներ: Ստացված թիվը գրել 'SUM' անունով նոր միջավայրի փոփոխականում: *Հուշում՝ օգտագործել սիմվոլային տողից թիվ և թվից սիմվոլային տող ստանալու համար նախատեսված ֆունկցիաներ (TCHAR տիպի համար _itot_s և _ttoi) կամ իրականացնել դրանք ինքնուրույն:*

5. Ծրագիր 3-ին նման եղանակով օգտագործել GetCurrentDirectory և SetCurrentDirectory ֆունկցիաները՝ կատարելով համապատասխան սխալների մշակումը: *Հուշում՝ զանգվածի չափի համար օգտագործել MAX_PATH հաստատունը:*

6. Փոփոխել PATH միջավայրի փոփոխականի արժեքը՝ ավելացնելով դրան նոր կատալոգ, և դուրս բերել PATH-ի նոր արժեքն էկրանին: PATH-ում ավելացվող կատալոգի անունը տրվում է հրամայական տողով:

ԳԼՈՒԽ 2
ՖԱՅԼԱՅԻՆ ՀԱՄԱԿԱՐԳԸ

Օպերացիոն համակարգի հետ աշխատանքում ծրագրավորողը սովորաբար առաջին հերթին կարիք է ունենում առնչվելու ֆայլային համակարգին և կոնսոլային մուտքի/ելքի գործողություններին: Ֆայլերը հիմնարար դեր են խաղում տվյալների երկարատև պահպանման և կառավարման հարցում, ինչպես նաև ապահովում են միջծրագրային համագործակցության պարզագույն ձևը: Այդ պատճառով, ինչպես բոլոր օպերացիոն համակարգերում, այնպես էլ «Windows»-ում ֆայլային համակարգը կենտրոնական բաղադրիչներից է, իսկ ֆայլը՝ հիմնական հասկացություններից մեկը:

Այս գլխում դիտարկվում են ֆայլերի, կատալոգների, ինչպես նաև մուտքի/ելքի ստանդարտ սարքերի հետ աշխատանքի սկզբունքները «Windows» օպերացիոն համակարգում, դիտարկվում են հիմնական համակարգային կանչերը:

2.1. Աշխատանքը ֆայլերի հետ

Ձեռնարկում դիտարկվում է «Windows»-ում օգտագործվող ֆայլային համակարգերից հիմնականը՝ NTFS-ը, որը ժամանակակից է, թույլատրում է ֆայլերի երկար անունները, ապահովում է անվտանգություն, վթարային իրավիճակների նկատմամբ կայունություն, ընդլայնված ատրիբուտներ և թույլ է տալիս աշխատել շատ մեծ ֆայլերի հետ: Նկատենք, որ բոլոր ֆայլային համակարգերի (FAT, CDFS, UFS և այլն) հասանելիությունը «Windows»-ն ապահովում է նույն ձևով, գուցե որոշակի սահմանափակումներով:

«Windows»-ն ապահովում է ֆայլերի անվանման հիերարխիկ համակարգ, որի համար գործում են հետևյալ կանոնները.

- Ֆայլի կամ կատալոգի լրիվ անունն սկսվում է արմատային կատալոգի անվանումից՝ C:, D: և այլն, իսկ որպես բաժանման սիմվոլ՝ օգտագործվում է \-ը: Ֆայլի լրիվ անվան օրինակ

է C:\Program Files (x86)\Microsoft Visual Studio\Installer\-setup.exe:

- Ֆայլի կամ կատալոգի անվան մեջ չի թույլատրվում օգտագործել որոշ հատուկ սիմվոլներ, ինչպիսիք են՝ < > : " | ? * \ /:
- Ֆայլի կամ կատալոգի անվան մեջ մեծատառ և փոքրատառ սիմվոլներն իրարից չեն տարբերվում: Օրինակ՝ C:\File.txt-ն և C:\file.txt-ը նույն ֆայլն են:
- Գոյություն ունի ֆայլի կամ կատալոգի անվան երկարության սահմանափակում, որը որոշվում է MAX_PATH թվով:
- Ֆայլի անվան մեջ մտնում է նաև ընդլայնումը, որը բաժանվում է սիմվոլի միջոցով և նկարագրում է ֆայլի ենթադրյալ տեսակը: Օրինակ՝ C:\prog.exe ֆայլի ընդլայնումն exe-ն է, ինչը սովորաբար նշանակում է, որ այն կատարվող ֆայլ է:
- Որպես կատալոգի անվանում՝ . և .. կրճատումներն օգտագործվում են համապատասխանաբար ընթացիկ կատալոգի և ընթացիկ կատալոգի հայրական կատալոգի անունների փոխարեն:

«Windows»-ում որևէ ֆայլի հետ ծրագրային աշխատանքի համար անհրաժեշտ է ունենալ այդ ֆայլի, այսպես կոչված, նկարագրիչը, որի մասին ավելի մանրամասն կխոսվի գլուխ 3-ում պրոցեսներն ուսումնասիրելիս: Նոր ֆայլեր ստեղծելու կամ գոյություն ունեցող ֆայլերը բացելու համար օգտագործվում է CreateFile ֆունկցիան (Աղ. 6): Վերջինս վերադարձնում է ֆայլի նկարագրիչը, որի հետ կատարվում է հետագա աշխատանքը: Ծրագրի աշխատանքի ընթացքում բացված բոլոր նկարագրիչների մասին ինֆորմացիան պահվում է հատուկ աղյուսակում (տե՛ս 3.2): Եթե որևէ պատճառով հնարավոր չի եղել ստեղծել նկարագրիչը, ֆունկցիան վերադարձնում է INVALID_HANDLE_VALUE թվային արժեքը: Դիտարկենք CreateFile ֆունկցիայի արգումենտներից որոշներն ավելի մանրամասն.

- dwDesiredAccess-ը թվային արժեք է, որով որոշվում է, թե ստեղծվող նկարագրիչի միջոցով ինչ գործողություններ են

թույլատրվում կատարել տվյալ ֆայլի հետ հետագայում: GENERIC_READ արժեքն օգտագործվում է կարդալու, իսկ GENERIC_WRITE-ը՝ գրելու հասանելիությունն ապահովելու համար: Կարելի է նաև համադրել այդ արժեքները բիթային «կամ»-ի միջոցով՝ GENERIC_READ | GENERIC_WRITE:

- dwShareMode-ը թվային արժեք է, որով որոշվում է, թե ինչ գործողություններ են թույլատրվում կատարել տվյալ ֆայլի հետ այլ պրոցեսների կողմից, քանի դեռ ստեղծվող նկարագրիչը փակված չէ: 0 արժեքն օգտագործվում է «կիսումը» բացարձակ արգելելու համար: Մնացած արժեքներին կարելի է ծանոթանալ «Microsoft»-ի պաշտոնական փաստաթղթերի էջում:
- lpSecAttr-ի միջոցով որոշվում են ստեղծվող նկարագրիչի պաշտպանական ատրիբուտները: NULL արժեքն օգտագործվում է լռությամբ պատշպանական ատրիբուտներ կցելու համար: Այլ արժեքներ փոխանցելու դեպքը կքննարկվի 3-րդ գլխում:
- dwCreationDisposition-ը թվային արժեք է, որով որոշվում են CreateFile ֆունկցիայի աշխատանքի առանձնահատկությունները: Օրինակ՝ OPEN_EXISTING արժեքն օգտագործվում է այն դեպքում, երբ տվյալ ֆայլն արդեն գոյություն ունի: Մնացած արժեքներին կարելի է ծանոթանալ «Microsoft»-ի պաշտոնական փաստաթղթերի էջում:
- dwFlagsAndAttributes-ը նույնպես թվային արժեք է, որով որոշվում են տվյալ ֆայլին տրվելիք հատկանիշները (դրոշակները և ատրիբուտները), եթե այդ ֆայլն ստեղծվում է CreateFile ֆունկցիայի միջոցով: Հնարավոր արժեքներին կարելի է ծանոթանալ «Microsoft»-ի պաշտոնական փաստաթղթերի էջում:

Ունենալով ֆայլի նկարագրիչը՝ կարող ենք կարդալ ֆայլից կամ գրել դրա մեջ: Ֆայլից կարդալու համար օգտագործվում են ReadFile,

իսկ ֆայլում գրելու համար՝ WriteFile համակարգային կանչերը (Աղ. 6)։ Ուշադրություն դարձնենք, որ ReadFile-ի առաջին արգումենտում տրվող նկարագրիչը պետք է ունենա առնվազն GENERIC_READ հասանելիություն, WriteFile-ի համապատասխան արգումենտը՝ առնվազն GENERIC_WRITE հասանելություն։

Որևէ բացված նկարագրիչ օգտագործելուց հետո այն պետք է փակել CloseHandle համակարգային կանչի միջոցով (Աղ. 6)։

Ֆունկցիա	Նկարագիր
HANDLE CreateFile(LPCSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ATTRIBUTES lpSecAttr, DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);	Ստեղծում և վերադարձնում է նկարագրիչ՝ lpFileName անունով ֆայլի համար։ Այդ նկարագրիչի միջոցով հետագայում կարելի է կարդալ կամ գրել ֆայլի մեջ՝ կախված dwDesiredAccess արգումենտի արժեքից։ Ֆունկցիան lpFileName անունով ֆայլը կարող է կա՛մ բացել, կա՛մ ստեղծել և նոր միայն բացել՝ կախված dwCreationDisposition արգումենտի արժեքից։ Եթե lpFileName անունով ֆայլն ստեղծվել է, ապա այն կունենա dwFlagsAndAttributes ատրիբուտները։
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);	hFile ֆայլային նկարագրիչի միջոցով համապատասխան ֆայլից կարդում է բայթեր lpBuffer-ի մեջ։ Ֆունկցիան փորձում է կարդալ nNumberOfBytesToRead քանակով բայթեր և lpNumberOfBytesRead արգումենտում գրում է, թե քանի բայթ ստացվեց կարդալ։

BOOL WriteFile(**HANDLE hFile,** **LPVOID lpBuffer,** **DWORD nNumberOfBytesToWrite,** **LPDWORD lpNumberOfBytesWritten,** **LPOVERLAPPED lpOverlapped** **);**	hFile ֆայլային նկարագրիչի միջոցով համապատասխան ֆայլում գրում է բայթեր lpBuffer-ից: Ֆունկցիան փորձում է գրել nNumberOfBytesToWrite քանակով բայթեր և lpNumberOfBytesWritten արգումենտում գրում է, թե քանի բայթ ստացվեց գրել:
BOOL CloseHandle(**HANDLE hObject** **);**	Փակում է hObject նկարագրիչը: Վերջինս կարող է լինել ինչպես ֆայլային նկարագրիչ, այնպես էլ կամայական այլ միջուկի օբյեկտի նկարագրիչ (տե՛ս 3.1):

Աղյուսակ 6: CreateFile, ReadFile, WriteFile և CloseHandle ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 4-ում, օգտագործելով Աղյուսակ 6-ի 4 ֆունկցիաները, իրականացված է copy_file ֆունկցիան, որն իր առաջին արգումենտում փոխանցվող անունով ֆայլի պարունակությունը պատճենում է երկրորդ արգումենտում փոխանցվող անունով ֆայլի մեջ: Ընդ որում, ենթադրվում է, որ առաջին ֆայլը գոյություն ունի, իսկ երկրորդը նոր է ստեղծվում (նոր ֆայլ է ստեղծվում, եթե նույնիսկ այդ անունով ֆայլ արդեն կար, իսկ վերջինս ջնջվում է):

copy_file ֆունկցիան իրականացված է հետևյալ քայլերի հերթականությամբ.

1. CreateFile ֆունկցիայի միջոցով ստեղծվում է նկարագրիչ name1 անունով ֆայլի համար: Նկարագրիչն ստեղծվում է կարդալու հասանելիությամբ՝ GENERIC_READ, և ենթադրությամբ, որ name1 ֆայլն արդեն գոյություն ունի՝ OPEN_EXISTING արժեքով dwCreationDisposition արգումենտի համար: Եթե նկարագրիչն ստեղծել չի ստացվում, ֆունկ-

ցիան դուրս է բերում հանդիպած սխալի տեքստը և վերադարձնում false:

2. CreateFile ֆունկցիայի միջոցով ստեղծվում է նկարագրիչ name2 անունով ֆայլի համար: Նկարագրիչն ստեղծվում է գրելու հասանելիությամբ՝ GENERIC_WRITE, և ենթադրությամբ, որ name2 ֆայլը պետք է ստեղծել՝ CREATE_ALWAYS արժեքով dwCreationDisposition արգումենտի համար: Եթե նկարագրիչն ստեղծել չի ստացվում, ֆունկցիան նախ փակում է name1-ի համար բացված նկարագրիչը, այնուհետև դուրս է բերում հանդիպած սխալի տեքստը և վերադարձնում false:
3. Այս կետում արդեն գոյություն ունեն երկու նկարագրիչներ՝ name1 անունով ֆայլի համար կարդալու հասանելիությամբ և name2 անունով ֆայլի համար գրելու հասանելիությամբ: Ֆունկցիան ReadFile և WriteFile ֆունկցիաների միջոցով առաջին ֆայլից կարդում է 10-ական բայթ և գրում այդ բայթերը երկրորդ ֆայլի մեջ այնքան ժամանակ, քանի դեռ առաջին ֆայլում բոլոր բայթերը կարդացված չեն:
4. copy_file ֆունկցիան աշխատանքն ավարտելուց առաջ փակում է երկու բացված նկարագրիչները և վերադարձնում true:

```
void error_text_output();

bool copy_file(const TCHAR* name1, const TCHAR* name2) {
        HANDLE h1 = CreateFile(name1,
                                GENERIC_READ,
                                0,
                                NULL,
                                OPEN_EXISTING,
                                0,
                                NULL);
        if (INVALID_HANDLE_VALUE == h1) {
```

```
            error_text_output();
            return false;
        }

        HANDLE h2 = CreateFile(name2,
                               GENERIC_WRITE,
                               0,
                               NULL,
                               CREATE_ALWAYS,
                               FILE_ATTRIBUTE_NORMAL,
                               NULL);
        if (INVALID_HANDLE_VALUE == h2) {
            CloseHandle(h1);
            error_text_output();
            return false;
        }

        const int s = 10;
        BYTE buffer[s];
        DWORD r = -1, w;
        while (ReadFile(h1, buffer, s, &r, NULL) && r != 0) {
            if (!WriteFile(h2, buffer, r, &w, NULL) || r != w) {
                CloseHandle(h1);
                CloseHandle(h2);
                error_text_output();
                return false;
            }
        }
        CloseHandle(h1);
        CloseHandle(h2);
        if (r != 0) {
            error_text_output();
            return false;
        }

        return true;
}
```

Ծրագիր 4։ Տրված ֆայլի պարունակության պատճենումը մեկ այլ ֆայլի մեջ։

Ծրագիր 4-ի իրականացման տրամաբանության հիմքում ընկած է այն, որ ReadFile ֆունկցիան ցիկլի ամեն իտերացիայի ժամանակ կարդում է ֆայլը ոչ թե սկզբից, այլ նախորդ իտերացիայում հասած տեղից: Նույն կերպ է վարվում նաև WriteFile ֆունկցիան: Այդ ֆունկցիների աշխատանքի համար այդպիսի վարք է ստացվում հետևյալ պատճառով. CreateFile ֆունկցիայի միջոցով ֆայլային նկարագրիչ ստեղծելիս վերջինիս հետ կապվում է ֆայլային ցուցիչ, որը ցույց է տալիս ընթացիկ բայթի համարը: Նոր ստեղծված ֆայլային նկարագրիչի համար ֆայլային ցուցչի արժեքը 0 է: ReadFile և WriteFile ֆունկցիաներն իրենց աշխատանքի ժամանակ փոփոխում են ֆայլային ցուցչի արժեքը՝ տեղափոխելով այն դեպի աջ՝ կարդացված կամ գրված բայթերի քանակին համապատասխան քանակով: Սակայն «Windows API»-ը տալիս է ֆունկցիա, որի միջոցով կարելի է փոփոխել ֆայլային ցուցչի արժեքը՝ SetFilePointer (Աղ. 7): Պետք է հաշվի առնել, որ ReadFile/WriteFile ֆունկցիաների աշխատանքի դեպի աջ «ուղղությունը» հնարավոր չէ փոխել:

Ֆայլային համակարգում յուրաքանչյուր ֆայլի հետ կապված են դրա հատկանիշները ներկայացնող մետատվյալներ (ատրիբուտներ): Դրանցից են ֆայլի չափը, ստեղծման, վերջին դիմումի, վերջին փոփոխության ժամանակները, թույլտվությունները (permission), դրոշակները (readonly, tmp, system, archive և այլն) և ուրիշներ: Մետատվյալների հետ աշխատանքի համար նախատեսված են հատուկ ֆունկցիաներ: Աղյուսակ 7-ում բերված են ֆայլի ժամանակները կարդալու՝ GetFileTime և փոփոխելու՝ SetFileTime ֆունկցիաները:

Ֆունկցիա	**Նկարագիր**
DWORD SetFilePointer(**HANDLE hFile,** **LONG lDistanceToMove,** **PLONG lpDistanceToMoveHigh,** **DWORD dwMoveMethod** **);**	Տեղաշարժում է hFile ֆայլային նկարագրիչի ֆայլային ցուցիչը՝ lDistanceToMove և lpDistanceToMoveHigh արգումենտներով փոխանցված 64-բիթանոց նշանով արժեքի բայթերի

	քանակով dwMoveMethod դիրքից: Վերջինս կարող է լինել FILE_BEGIN՝ ֆայլի սկիզբ, FILE_CURRENT՝ ֆայլային ցուցչի ընթացիկ դիրք, կամ FILE_END՝ ֆայլի վերջ:
BOOL GetFileTime(**HANDLE hFile,** **LPFILETIME lpCreationTime,** **LPFILETIME lpLastAccessTime,** **LPFILETIME lpLastWriteTime** **);**	Գրում է hFile ֆայլային նկարագրիչին համապատասխանող ֆայլի ստեղծման, վերջին դիմումի և վերջին փոփոխության ժամանակները համապատասխանաբար lpCreationTime, lpLastAccessTime և lpLastWriteTime արգումենտներում:
BOOL SetFileTime(**HANDLE hFile,** **const FILETIME** ***lpCreationTime,** **const FILETIME** ***lpLastAccessTime,** **const FILETIME** ***lpLastWriteTime** **);**	Փոխում է hFile ֆայլային նկարագրիչին համապատասխանող ֆայլի ստեղծման, վերջին դիմումի և վերջին փոփոխության ժամանակները համապատասխանաբար lpCreationTime-ով, lpLastAccessTime-ով և lpLastWriteTime-ով, եթե դրանք 0-ական ցուցիչ չեն:

Աղյուսակ 7: SetFilePointer, GetFileTime և SetFileTime ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 5-ում, օգտագործելով Աղյուսակ 7-ում ներկայացված GetFileTime և SetFileTime ֆունկցիաները, իրականացված է change_file_creation_date ֆունկցիան, որն իր արգումենտում փոխանցվող անունով ֆայլի ստեղծման ժամանակը փոխում է ներկա պահի արժեքով:

```
bool change_file_creation_date(const TCHAR* name) {
	SetLastError(0);
	HANDLE h = CreateFile(name,
				GENERIC_WRITE,
				0,
				NULL,
				OPEN_ALWAYS,
				FILE_ATTRIBUTE_NORMAL,
				NULL);
	if (INVALID_HANDLE_VALUE == h) {
		error_text_output();
		return false;
	}
	if (ERROR_ALREADY_EXISTS != GetLastError()) {
		CloseHandle(h);
		return true;
	}
	FILETIME creationTime{};
	SYSTEMTIME currentTime{};
	GetSystemTime(&currentTime);
	if (!FileTimeToSystemTime(&creationTime, &currentTime)) {
		CloseHandle(h);
		return false;
	}
	if (!SetFileTime(h, &creationTime, NULL, NULL)) {
		CloseHandle(h);
		return false;
	}
	CloseHandle(h);
	return true;
}
```

Ծրագիր 5: Տրված ֆայլի ստեղծման ամսաթվի փոփոխումը:

2.2. Աշխատանքը կոնսոլի և մուտքի/ելքի ստանդարտ սարքերի հետ

Կամայական «Windows» կիրառության հետ կապված է 0 կամ 1 հատ հատուկ տեքստային ֆայլ, որը կոչվում է կոնսոլ: Եթե կիրառության տեսակը կոնսոլային է (CUI – Console User Interface), ապա դրան լռությամբ հատկացվում է կոնսոլ, հակառակ դեպքում (GUI – Graphical User Interface)՝ ոչ: Սակայն կիրառության աշխատանքի ընթացքում կարելի է դրան հատկացնել կոնսոլ, եթե այն արդեն հատկացված չէ, կամ ազատել կոնսոլը, եթե այն արդեն հատկացված է՝ օգտագործելով համապատասխանաբար AllocConsole և FreeConsole համակարգային կանչերը (Աղ. 8):

Կոնսոլից կարդալու և կոնսոլում գրելու համար նկարագրիչ է ստեղծվում CreateFile ֆունկցիայի կանչի միջոցով՝ որպես ֆայլի անուն փոխանցելով համապատասխանաբար “CONIN$” և “CONOUT$” սիմվոլային տողերը:

Կոնսոլից կարդալու և կոնսոլում գրելու համար կարելի է օգտագործել ինչպես արդեն ծանոթ ReadFile և WriteFile, այնպես էլ ReadConsole և WriteConsole ֆունկցիաները (Աղ. 8): Վերջիններս տարբերվում են միայն նրանով, որ աշխատում են ոչ թե բայթերով, այլ սիմվոլներով:

Ֆունկցիա	Նկարագիր
BOOL AllocConsole();	Կանչող կիրառությանը կոնսոլ է հատկացնում:
BOOL FreeConsole();	Կանչող կիրառության կոնսոլն ազատում է:
BOOL ReadConsole(HANDLE hConsoleInput, LPVOID lpBuffer, DWORD nNumberOfCharsToRead, LPDWORD	hConsoleInput կոնսոլի նկարագրիչի միջոցով կոնսոլից սիմվոլներ է կարդում lpBuffer-ի մեջ: Ֆունկցիան փորձում է կարդալ nNumberOfCharsToRead քանակով սիմվոլներ և

lpNumberOfCharsRead, **LPVOID pInputControl** **);**	lpNumberOfCharsRead արգումենտում գրում է, թե քանի սիմվոլ ստացվեց կարդալ:
BOOL WriteConsole(**HANDLE hConsoleOutput,** **const VOID *lpBuffer,** **DWORD** **nNumberOfCharsToWrite,** **LPDWORD** **lpNumberOfCharsWritten,** **LPVOID lpReserved** **);**	hConsoleOutput կոնսոլի նկարագրիչի միջոցով կոնսոլում սիմվոլներ է գրում lpBuffer-ից: Ֆունկցիան փորձում է գրել nNumberOfCharsToWrite քանակով սիմվոլներ և lpNumberOfCharsWritten արգումենտում գրում է, թե քանի սիմվոլ ստացվեց գրել:

Աղյուսակ 8: AllocConsole, FreeConsole, ReadConsole և WriteConsole ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Կամայական Windows կիրառության համար նախատեսված են երեք ստանդարտ սարքեր՝ տվյալների մուտքի, տվյալերի ելքի և սխալների հաղորդագրությունների դուրսբերման համար: Այս երեք ստանդարտ սարքերի նկարագրիչներն ստեղծվում են կիրառության աշխատանքն սկսելիս և ավելացվում նկարագրիչների աղյուսակին: Հետևաբար ստանդարտ սարքերի հետ աշխատելու համար դրանց նկարագրիչները պետք է ոչ թե ստեղծել, այլ ուղղակի վերցնել GetStdHandle համակարգային կանչի օգնությամբ (Աղ. 9): Ստանդարտ սարքերի նկարագրիչները CloseHandle ֆունկցիայի միջոցով փակելու անհրաժեշտություն չկա (դրանցից որևէ մեկի փակելը կբերի համապատասխան սարքի անհասանելի դառնալուն):

Ֆունկցիա	**Նկարագիր**
HANDLE GetStdHandle(**DWORD nStdHandle** **);**	Վերադարձնում է nStdHandle թվային արժեքին համապատասխանող ստանդարտ սարքի նկարագրիչը: nStdHandle-ի արժեքը կարող է լինել

	STD_INPUT_HANDLE (ստանդարտ մուտք), STD_OUTPUT_HANDLE (ստանդարտ ելք) և STD_ERROR_HANDLE (սխալների հաղորդագրությունների դուրսբերում) հաստատուններից որևէ մեկը:
BOOL SetStdHandle(**DWORD nStdHandle,** **HANDLE hHandle** **);**	nStdHandle թվային արժեքին համապատասխանող սարքը կապում է hHandle նկարագրիչի հետ:

Աղյուսակ 9: GetStdHandle և SetStdHandle ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

«Windows» կիրառությունների համար երեք ստանդարտ սարքերը լռությամբ կապված են կոնսոլի հետ: Սակայն դա կարելի է փոխել՝ ստանդարտ սարքը որևէ այլ նկարագրիչի հետ կապելով: Դրա համար օգտագործվում է SetStdHandle համակարգային կանչը (Աղ. 9): Տրամաբանական է, որ ստանդարտ մուտքի հետ կապվող նկարագրիչը պետք է ունենա առնվազն GENERIC_READ հասանելիություն, իսկ ստանդարտ ելքի և սխալների հաղորդագրությունների դուրսբերման սարքերի հետ կապվողը՝ առնվազն GENERIC_WRITE հասանելիություն:

Ծրագիր 6-ում, օգտագործելով Աղյուսակ 9-ի 2 ֆունկցիաները, իրականացված է copy_file_to_std_output ֆունկցիան, որն իր արգումենտում փոխանցվող անունով ֆայլի պարունակությունը պատճենում է ստանդարտ ելքի մեջ: main ֆունկցիայում նախ կանչվում է copy_file_to_std_output ֆունկցիան՝ ցուցադրելու համար, որ լռությամբ ստանդարտ ելքի սարքը կապված է կոնսոլի հետ: Այնուհետև ստանդարտ ելքի սարքը կապվում է մի ֆայլի նկարագրիչի հետ, որի անունն ստացվում է հրամայական տողով: copy_file_to_std_output ֆունկցիայի հաջորդ կանչի արդյունքից պարզ է դառնում, որ ստանդարտ ելքի սարքն այլևս կոնսոլը չէ, այլ նշված ֆայլն է: Ակնհայտ է նաև, որ C-ական printf/scanf ֆունկցիաները, ինչպես նաև C++-ական

std::cout/std::cin օբյեկտներն աշխատում են կոնսոլի հետ՝ անկախ ստանդարտ սարքերից։

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

bool copy_file_to_std_output(const TCHAR* name) {
	HANDLE hFile = CreateFile(name,
				GENERIC_READ,
				0,
				NULL,
				OPEN_EXISTING,
				0,
				NULL);
	if (INVALID_HANDLE_VALUE == hFile) {
		error_text_output();
		return false;
	}

	HANDLE hOutput = GetStdHandle(STD_OUTPUT_HANDLE);
	if (INVALID_HANDLE_VALUE == hOutput) {
		CloseHandle(hFile);
		error_text_output();
		return false;
	}

	const int s = 10;
	BYTE buffer[s];
	DWORD r = -1, w;
	while (ReadFile(hFile, buffer, s, &r, NULL) && r != 0) {
		if (!WriteFile(hOutput, buffer, r, &w, NULL) || r != w) {
			CloseHandle(hFile);
			error_text_output();
			return false;
```

```
                }
        }
        CloseHandle(hFile);
        if (r != 0) {
                error_text_output();
                return false;
        }

        return true;
}

int _tmain(int argc, TCHAR* argv[]) {
        if (argc > 3) {
                // output will be on console
                _tprintf(_T("Console output before change of std
output!\n"));
                // output will be on console
                copy_file_to_std_output(argv[1]);

                // Change std output to a file
                HANDLE hFile = CreateFile(argv[2],
                                        GENERIC_WRITE,
                                        0,
                                        NULL,
                                        CREATE_ALWAYS,
                                        FILE_ATTRIBUTE_NORMAL,
                                        NULL);
                if (INVALID_HANDLE_VALUE == hFile) {
                        error_text_output();
                        return 1;
                }
                if (!SetStdHandle(STD_OUTPUT_HANDLE, hFile)) {
                        error_text_output();
                        return 2;
                }
                // output will be on console
                _tprintf(_T("\nConsole output after change of std
output!\n"));
                // output will be in file
                copy_file_to_std_output(argv[1]);
```

```
                // Get handle of console output
                HANDLE hConsole = CreateFile(_T("CONOUT$"),
                                                GENERIC_WRITE,
                                                0,
                                                NULL,
                                                OPEN_EXISTING,
                                                0,
                                                NULL);
                if (INVALID_HANDLE_VALUE == hConsole) {
                        error_text_output();
                        return 3;
                }
                TCHAR buffer[] = _T("TEST TEXT\n");
                // Output will be on console
                if (!WriteFile(hConsole, buffer, 11, NULL, NULL)) {
                        error_text_output();
                        return 4;
                }
        }

        return 0;
}
```

Ծրագիր 6: Ստանդարտ սարքերի հետ աշխատանքը:

2.3. Լրացուցիչ ֆունկցիաներ

Դիտարկենք ֆայլերի հետ աշխատանքի համար «Windows» օպերացիոն համակարգի կողմից տրամադրվող հետևյալ ֆունկցիաները ևս (Աղ. 10):

Ֆունկցիա	Նկարագիր
BOOL DeleteFile(LPCSTR lpFileName);	Ջնջում է lpFileName անունով ֆայլը:
BOOL CopyFile(	Պատճենում է

LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName, BOOL bFailIfExists);	lpExistingFileName անունով ֆայլի պարունակությունը lpNewFileName անունով ֆայլի մեջ: bFailIfExists արժեքով որոշվում է՝ 2-րդ ֆայլը պետք է ստեղծել, թե ոչ:
BOOL MoveFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName);	lpExistingFileName անունով ֆայլը տեղափոխում է՝ ձեռք բերելով lpNewFileName անունը:
BOOL GetFileSizeEx(HANDLE hFile, PLARGE_INTEGER lpFileSize);	hFile նկարագրիչով ֆայլի չափը բայթերով գրում է lpFileSize արգումենտում:
BOOL GetFileInformationByHandle(HANDLE hFile, LPBY_HANDLE_FILE_INFORMATION lpFI);	hFile նկարագրիչով ֆայլի մասին ինֆորմացիան գրում է lpFI արգումենտում:

Աղյուսակ 10: DeleteFile, CopyFile, MoveFile, GetFileSizeEx և GetFileInformationByHandle ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

2.4. Աշխատանքը կատալոգների հետ

Ֆայլային համակարգում էական դեր են խաղում կատալոգները: Կատալոգն իրականացման տեսանկյունից նույնպես ֆայլ է, որի մեջ գրված են այն ֆայլերի ու ենթակատալոգների անունները և այլ ինֆորմացիա, որոնք պարունակվում են այդ կատալոգում: Կատալոգ հանդիսացող ֆայլն ունի հատուկ ատրիբուտ՝ FILE_ATTRIBUTE_DIRECTORY:

Կատալոգների հետ կապված ստանդարտ գործողություններ են կատալոգի ստեղծումը, կատալոգի հեռացումը ֆայլային համակար-

գից և կատալոգի ընթերցումը (շրջանցումը, այսինքն՝ հերթով դիտարկվում է պարունակությունը)։ Կատալոգ ստեղծելու համար Windows-ը տրամադրում է CreateDirectory, իսկ հեռացնելու համար՝ RemoveDirectory ֆունկցիան (Աղ. 11)։

Ֆունկցիա	Նկարագիր
BOOL CreateDirectory(LPCSTR lpPathName, LPSECURITY_ATTRIBUTES lpSecAttr);	Ստեղծում է lpPathName անունով կատալոգ։
BOOL RemoveDirectory(LPCSTR lpPathName);	Ջնջում է lpPathName անունով կատալոգը, եթե այն դատարկ է։

Աղյուսակ 11։ CreateDirectory և RemoveDirectory ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը։

Կատալոգի ընթերցումը կատարվում է 3 ֆունկցիաների համատեղ օգտագործմամբ՝ FindFirstFile, FindNextFile և FindClose։ Առաջինով տրվում է ընթերցելու համար անհրաժեշտ կատալոգը։ Ֆունկցիան վերադարձնում է ֆայլային նկարագրիչ, որը հետագայում օգտագործվում է FindNextFile ֆունկցիայի կանչի ժամանակ։ Վերջինս անցնում է FindFirstFile-ին տրված կատալոգի հաջորդ ֆայլին։ FindFirstFile-ի վերադարձրած ֆայլային նկարագրիչը կարելի է փակել միայն FindClose ֆունկցիայի միջոցով (Աղ. 12)։

Ֆունկցիա	Նկարագիր
HANDLE FindFirstFile(LPCSTR lpFileName, LPWIN32_FIND_DATA lpFindFileData);	Որոնում է lpFileName արժեքին համապատասխան անունով ֆայլ կամ կատալոգ և վերջինիս մասին ինֆորմացիան գրում է lpFindFileData արգումենտում։ Վերադարձնում է որոնման նկարագրիչը։

BOOL FindNextFile(**HANDLE hFindFile,** **LPWIN32_FIND_DATA lpFindFileData** **);**	Անցնում է hFindFile որոնման նկարագրիչին համապատասխան հաջորդ ֆայլին կամ կատալոգին և վերջինիս մասին ինֆորմացիան գրում է lpFindFileData արգումենտում:
BOOL FindClose(**HANDLE hFindFile** **);**	Փակում է hFindFile որոնման նկարագրիչը:

Աղյուսակ 12: FindFirstFile, FindNextFile և FindClose ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 7-ում իրականացված է traverse_current_directory ֆունկցիան, որը շրջանցում է ընթացիկ կատալոգը և դրանում պարունակվող ֆայլերի, կատալոգների անունները, չափերն ու ստեղծման ամսաթվերը դուրս է բերում կոնսոլում:

```
bool traverse_current_folder() {
        SetLastError(0);
        WIN32_FIND_DATA data;
        HANDLE h = FindFirstFile(_T("*"), &data);
        if (INVALID_HANDLE_VALUE == h) {
                error_text_output();
                return false;
        }

        SYSTEMTIME st;
        do {
                FileTimeToSystemTime(&data.ftCreationTime, &st);
                _tprintf(_T("Name: %s, Size %d, Date: %d/%d/%d\n"),
                 data.cFileName,
                        (data.nFileSizeHigh * MAXDWORD) +
                data.nFileSizeLow,
                        st.wDay,
                        st.wMonth,
                        st.wYear);
```

```
        } while (FindNextFile(h, &data));

        FindClose(h);

        if (GetLastError() != ERROR_NO_MORE_FILES) {
                error_text_output();
                return false;
        }

        return true;
}
```

Ծրագիր 7: Ընթացիկ կատալոգի շրջանցումը:

Ծրագիր 8-ում իրականացված է traverse_sub_directories ֆունկցիան, որը կոնսոլում դուրս է բերում ընթացիկ կատալոգի այն ենթակատալոգների անունները, որոնք չեն պարունակում ենթակատալոգներ: Իրականացված է 2 օգնող ֆունկցիա՝ is_directory, որն ստանում է ֆայլի ատրիբուտներն ու որոշում՝ այն կատալոգ է, թե ոչ, և contains_sub_directory, որն ստանում է կատալոգի անունն ու որոշում՝ այն պարունակում է ենթակատալոգներ, թե ոչ:

```
bool is_directory(DWORD attributes) {
        return (attributes & FILE_ATTRIBUTE_DIRECTORY) != 0;
}

bool contains_sub_directory(const TCHAR* directory_name) {
        TCHAR name[MAX_PATH];
        _tcscpy_s(name, directory_name);
        _tcscat_s(name, _T("\\\\*"));

        WIN32_FIND_DATA data;
        HANDLE h = FindFirstFile(name, &data);
        if (INVALID_HANDLE_VALUE == h) {
                output_error_text();
                return false;
```

```
	}

	do {
		if (is_catalog(data) &&
		 _tcscmp(data.cFileName, _T(".")) != 0 &&
		 _tcscmp(data.cFileName, _T("..")) != 0) {
			FindClose(h);
			return true;
		}
	} while (FindNextFile(h, &data));

	FindClose(h);

	return true;
}

bool traverse_sub_folders() {
	SetLastError(0);
	WIN32_FIND_DATA data;
	HANDLE h = FindFirstFile(_T("*"), &data);
	if (INVALID_HANDLE_VALUE == h) {
		error_text_output();
		return false;
	}

	do {
		if (is_directory(data.dwFileAttributes) &&
		 !contains_sub_directory(data.cFileName)) {
			_tprintf(_T("Name: %s\n"), data.cFileName);
		}
	} while (FindNextFile(h, &data));

	FindClose(h);

	if (GetLastError() != ERROR_NO_MORE_FILES) {
		error_text_output();
		return false;
	}
```

```
        return true;
}
```

Ծրագիր 8։ Ենթակատալոգներ չպարունակող կատալոգների անունների դուրսբերումը։

Խնդիրներ

1. Գրել ֆունկցիա, որը name1 անունով ֆայլի պարունակությունը պատճենում է name2 անունով ֆայլի մեջ հակառակ կարգով։ Ֆայլերի անուններն ստանալ հրամայական տողով։

2. Գրել ֆունկցիա, որը name1 անունով ֆայլի առաջին n բայթը (բառը) տողը պատճենում է name2 անունով ֆայլի մեջ։ Ֆայլերի անուններն ստանալ հրամայական տողով։

3. Գրել ֆունկցիա, որը name1 անունով ֆայլի վերջին n բայթը (բառը) տողը պատճենում է name2 անունով ֆայլի մեջ։ Ֆայլերի անուններն ստանալ հրամայական տողով։

4. Գրել ֆունկցիա, որը ստանդարտ մուտքի պարունակությունը պատճենում է name1 անունով ֆայլի մեջ։ Ստանդարտ մուտքը փոխել name2 անունով ֆայլով։ Ֆայլերի անուններն ստանալ հրամայական տողով։

5. Գրել ֆունկցիա, որը ստանդարտ մուտքի պարունակությունը պատճենում է ստանդարտ ելքի մեջ։ Ստանդարտ մուտքը փոխել name1, իսկ ստանդարտ ելքը՝ name2 անունով ֆայլերով։ Ֆայլերի անուններն ստանալ հրամայական տողով։

6. Գրել ծրագիր, որը պատճենում է ֆայլի պարունակությունը կոնսոլում և ընդհատում է այդ գործողությունը CTRL+C ազդանշանով։ Ֆայլի անունն ստանալ հրամայական տողով։ *Հուշում՝ ուսումնասիրել SetConsoleCtrlHandler ֆունկցիան։*

7. Ստեղծել ֆայլ, որը կպարունակի միջավայրի բոլոր փոփոխականների անուններն ու արժեքները։ Ֆայլի անունն ստանալ հրամայական տողով։ Եթե հրամայական տողում ֆայլի անուն չկա, ապա

միջավայրի փոփոխականների անուններն ու արժեքները դուրս բերել կոնսոլում:

8. Գրել ծրագիր, որը կոնսոլում դուրս է բերում հրամայական տողով տրված ֆայլերի վերջին n տողերը:

9. Գրել ծրագիր, որը հաշվում է հրամայական տողով տրված տեքստային ֆայլերում բառերի քանակները: Ստանդարտ ելքում դուրս բերել ընթացիկ կատալոգի անունը, ֆայլերի անուններն ու յուրաքանչյուր ֆայլի համար բառերի քանակը:

10. Փոխել ընթացիկ կատալոգը dir-ով և սկզբնական ընթացիկ կատալոգում գտնվող old.txt ֆայլի պարունակությունը պատճենել dir կատալոգում ստեղծվող new.txt ֆայլի մեջ: dir կատալոգի անունն ստանալ հրամայական տողով:

11. Կոնսոլում դուրս բերել ընթացիկ կատալոգի բոլոր ենթակատալոգների անունները: Այնուհետև ծնողական կատալոգը դարձնել ընթացիկ: Կրկնել նկարագրված գործողությունն այնքան, մինչև որ արմատային կատալոգը դառնա ընթացիկ:

12. d:\dir կատալոգում ստեղծել text.txt ֆայլը, որը կպարունակի հրամայական տողով տրված ֆայլերի պարունակությունների միավորումը:

13. dir կատալոգում գտնել այն ենթակատալոգը, որն ստեղծվել է վերջում (ըստ ամսաթվի), և կոնսոլում դուրս բերել դրա պարունակած բոլոր ֆայլերի անունները: dir կատալոգի անունն ստանալ հրամայական տողով: *Հուշում՝ կարելի է օգտագործել ստանդարտ գրադարանի վեկտոր դասն ու տեսակավորման ալգորիթմը՝ գրելով տեսակավորման հայտանիշ:*

14. Կոնսոլում դուրս բերել հրամայական տողով տրված կատալոգի ֆայլերի վերջին n տողերը: Եթե հրամայական տողում կատալոգի անուն չկա, ապա դիտարկել ընթացիկ կատալոգը:

15. Ստեղծել text.txt ֆայլը, որը կպարունակի հրամայական տողով տրված կատալոգի բոլոր տեքստային ֆայլերի պարունակու-

թյունների միավորումը: Եթե հրամայական տողում կատալոգի անուն չկա, ապա դիտարկել ընթացիկ կատալոգը:

16. Գրել ծրագիր, որը կոնսոլում դուրս է բերում ընթացիկ կատալոգի պարունակությունը ֆայլերի չափերի աճման կարգով: Ընդ որում, կատալոգների անունները դուրս բերել առաջին հերթին: *Հուշում՝ կարելի է օգտագործել ստանդարտ գրադարանի վեկտոր դասն ու տեսակավորման ալգորիթմը՝ գրելով տեսակավորման հայտանիշ:*

17. Գրել ծրագիր, որը կոնսոլում դուրս է բերում ընթացիկ կատալոգի պարունակությունն անունների այբբենական կարգով: Ընդ որում, դուրս չբերել կատալոգների անունները: *Հուշում՝ կարելի է օգտագործել ստանդարտ գրադարանի վեկտոր դասն ու տեսակավորման ալգորիթմը՝ գրելով տեսակավորման հայտանիշ:*

ԳԼՈՒԽ 3
ՊՐՈՑԵՍՆԵՐԸ ԵՎ ՀՈՍՔԵՐԸ

Օպերացիոն համակարգերի հիմնական հասկացություններից են նաև պրոցեսը և հոսքը, որոնց միջոցով իրականացվում է բազմախնդիր աշխատանքը (multitasking)՝ միաժամանակ բազմաթիվ տարբեր ծրագրերի կատարումը։

Այս գլխում տրվում է նախնական պատկերացում «Windows» օպերացիոն համակարգում միջուկի օբյեկտների, այնուհետև՝ Windows-ական պրոցեսների ստեղծման, ավարտի և ղեկավարման մասին։ Պրոցեսների հետ աշխատանքից բացի՝ դիտարկվում է նաև Windows-ական հոսքի գաղափարը՝ դրա ստեղծումը, ավարտը և ղեկավարումը։

3.1. Միջուկի օբյեկտները

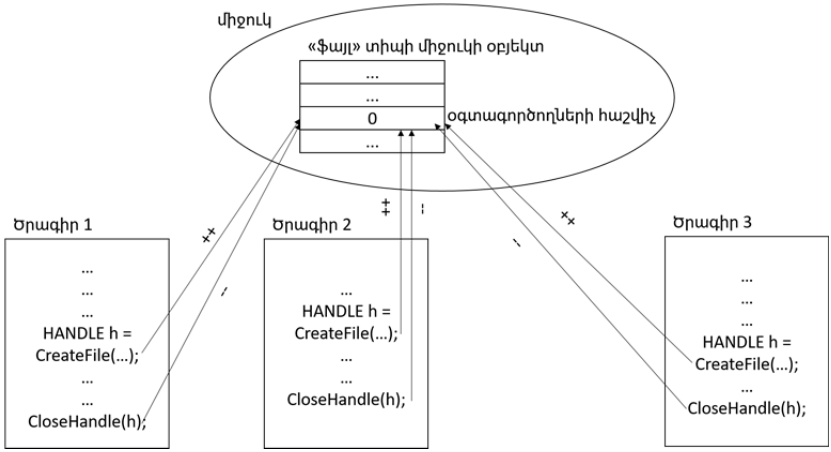
Ֆայլերի և կատալոգների հետ աշխատանքի ընթացքում ակնհայտ է դառնում, որ «Windows API»-ի ֆունկցիաների մեծ մասն աշխատում է նկարագրիչների (HANDLE տիպի) հետ, որոնք մինչ այժմ հանդիսանում էին ֆայլային նկարագրիչներ։ Սակայն ընդհանուր դեպքում նկարագրիչները «նկարագրում են» ոչ միայն ֆայլեր, այլ տարբեր տիպի միջուկի օբյեկտներ։ Վերջիններս օգտագործվում են օպերացիոն համակարգի կողմից տարբեր ռեսուրսներ ղեկավարելու համար՝ ֆայլեր (տե՛ս 2.1), պրոցեսներ, հոսքեր (տե՛ս՝ 3.1, 3.3), մյուտեքսներ, սեմաֆորներ (տե՛ս 4.2) և այլն։

Միջուկի օբյեկտը հիշողության բլոկ է, որը հատկացվում է օպերացիոն համակարգի միջուկի կողմից և հասանելի է միայն վերջինիս։ Հիշողության այդ բլոկն օբյեկտի մասին ինֆորմացիա պարունակող տվյալների կառույց է։ Որոշ դաշտեր ընդհանուր են միջուկի բոլոր օբյեկտների համար (օրինակ՝ օգտագործողների հաշվիչը, որի մասին կխոսվի ներքևում), իսկ որոշ դաշտներն էլ հատուկ են կոնկ-

րետ տիպի միջուկի օբյեկտների համար։ Օրինակ՝ «ֆայլ» տիպի միջուկի օբյեկտի ներքին կառուցվածքը մեծապես տարբերվում է «պրոցես» տիպի միջուկի օբյեկտի ներքին կառուցվածքից։

Քանի որ միջուկի օբյեկտների կառուցվածքը հասանելի է միայն միջուկին, ոչ մի կիրառություն հնարավորություն չունի գտնելու կամ փոփոխելու որևէ օբյեկտի պարունակությունը։ Այդ պատճառով «Windows API»-ը տրամադրում է ֆունկցիաներ, որոնց կանչի ժամանակ օպերացիոն համակարգն ստեղծում կամ գտնում է անհրաժեշտ միջուկի օբյեկտը և վերադարձնում դրա նկարագրիչը։ Վերջինս արդեն հնարավոր է փոխանցել «Windows API»-ի այլ ֆունկցիաների, որոնք աշխատում են համապատասխան միջուկի օբյեկտի հետ։

Այժմ դիտարկենք օգտագործողների հաշվիչի գաղափարը, որը, ինչպես նշվեց, ընդհանուր է բոլոր միջուկի օբյեկտների համար։ Միջուկի օբյեկտներն ստեղծել կամ ջնջել կարող է միայն օպերացիոն համակարգի միջուկը։ Սակայն նույն միջուկի օբյեկտի համար տարբեր կիրառություններ կարող են նկարագրիչներ բացել։ Այդ պատճառով միջուկի բոլոր օբյեկտներն ունեն թվային դաշտ՝ օգտագործողների հաշվիչ, որտեղ պահվում է այդ օբյեկտի համար բացված նկարագրիչների քանակը (օգտագործողների քանակը)։ Օբյեկտի ստեղծման ժամանակ այդ հաշվիչին տրվում է 1 արժեքը։ Երբ տվյալ օբյեկտի համար բացվում է ևս մի նկարագրիչ, հաշվիչի արժեքն ավելանում է 1-ով։ Բնականաբար, նկարագրիչը փակվելիս (օրինակ՝ CloseHandle ֆունկցիայի կանչի միջոցով) այդ հաշվիչի արժեքը պակասում է 1-ով։ Երբ օբյեկտի օգտագործողների հաշվիչի արժեքը դառնում է 0, միջուկը ջնջում է հիշողությունից «անպետք» օբյեկտը (Նկ. 2)։



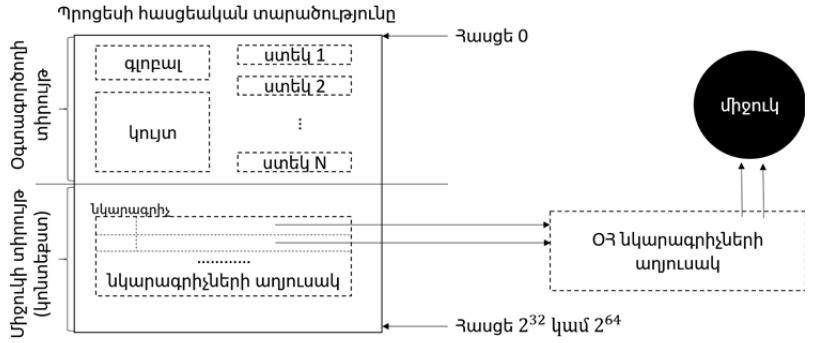
Նկար 2: «Ֆայլ» միջուկի օբյեկտի օգտագործողների հաշվիչի արժեքի ածը (++) և նվազումը (--) համապատասխանաբար CreateFile և CloseHandle ֆունկցիաների կանչերի ժամանակ:

3.2. Աշխատանքը պրոցեսների հետ

Պրոցեսը սահմանվում է որպես կատարվող ծրագրի (կիրառության) մեկ օրինակ: «Windows» օպերացիոն համակարգում պրոցեսը որոշվում է հետևյալ 2 բաղադրիչներով.

1. «Պրոցես» տիպի միջուկի օբյեկտ, որի միջոցով օպերացիոն համակարգը ղեկավարում է պրոցեսը: Այս միջուկի օբյեկտն օժտված է թվային նույնարկիչով (իդենտիֆիկատոր), որը միարժեքորեն որոշում է տվյալ պրոցեսը օպերացիոն համակարգում աշխատող բոլոր պրոցեսների բազմության մեջ:

2. Հասցեական տարածություն, որը պարունակում է բոլոր մոդուլների կոդերն ու տվյալները: Հենց այստեղ են հատկացվում հիշողության գլոբալ, դինամիկ բաշխման (կույտ) և հոսքերի լոկալ ստեկերի տիրույթները (Նկ. 3):



Նկար 3: Պրոցեսի հասցեական տարածությունը:

Նկատենք, որ պրոցեսներն իներտ են. հրամանների որևէ հաջորդականություն կատարում են հոսքերը (տե՛ս 3.3)։ Այդ պատճառով յուրաքանչյուր պրոցես ունի գոնե 1 հոսք՝ մուտքային, առաջնային կամ գլխավոր, որի կոդը գտնվում է հայտնի main ֆունկցիայի մարմնում։

Ինչպես երևում է Նկար 3-ից, պրոցեսի հասցեական տարածությունը կարելի է պայմանականորեն բաժանել 2 մասի՝ օգտագործողի տիրույթ և միջուկի տիրույթ։ Միջուկի տիրույթում է գտնվում, այսպես կոչված, նկարագրիչների աղյուսակը, որում գրանցվում են պրոցեսի կատարման ընթացքում բացված բոլոր նկարագրիչները։ HANDLE տիպի փոփոխականը, փաստորեն, այդ աղյուսակի համապատասխան տողի վրա ցուցիչ է։ Պրոցեսի ավարտի ժամանակ փակվում են նկարագրիչների աղյուսակի բոլոր նկարագրիչները։

Պրոցեսի ստեղծումը: «Windows» օպերացիոն համակարգում նոր պրոցես ստեղծելու համար օգտագործվում է CreateProcess համակարգային կանչը (Աղ. 13)։ Ընդունված է CreateProcess կանչող պրոցեսն անվանել ծնողական (հայրական), իսկ ստեղծվող պրոցեսը՝ որդիական։ Դիտարկենք այս ֆունկցիայի որոշ արգումենտներն ավելի մանրամասնորեն.

- lpApplicationName սիմվոլային տողով փոխանցվում է կիրառության անունը (կատարվող ֆայլի անունը), որը պետք է գործարկել: Այս արգումենտի արժեքը կարելի է NULL փոխանցել, իսկ կատարվող ֆայլի անունը՝ փոխանցել երկրորդ արգումենտում:
- lpCommandLine սիմվոլային տողով ստեղծվող պրոցեսին փոխանցվում է հրամայական տողը:
- bInheritHandles տրամաբանական արժեքով որոշվում է՝ արդյոք ծնողական պրոցեսի նկարագրիչների աղյուսակը ժառանգվում է ստեղծվող պրոցեսին, թե ոչ:
- dwCreationFlags թվային արժեքով որոշվում են ստեղծվող պրոցեսի որոշակի առանձնահատկություններ: Եթե հատուկ անհրաժեշտություն չկա, ապա այս արգումենտի արժեքը փոխանցվում է 0: Այլ արժեքներին կարելի է ծանոթանալ «Microsoft»-ի պաշտոնական փաստաթղթերի էջում:
- lpEnvironment-ն ստեղծվող պրոցեսին փոխանցվող միջավայրի փոփոխականների բլոկի վրա ցուցիչ է: NULL արժեքը փոխանցելով՝ որդիական պրոցեսին ժառանգվում է ծնողական պրոցեսի միջավայրի փոփոխականների բլոկը:
- lpCurrentDirectory-ով որոշվում է ստեղծվող պրոցեսի ընթացիկ կատալոգը: NULL արժեքը փոխանցելով՝ որդիական պրոցեսի ընթացիկ կատալոգը նույնն է, ինչ ծնողականինը:
- lpProcessInfo PROCESS_INFORMATION տիպի արգումենտում գրվում են ստեղծված պրոցեսի և վերջինիս առաջնային հոսքի նկարագրիչների ու նույնարկիչների արժեքները:

Ֆունկցիա	**Նկարագիր**
BOOL CreateProcess(**LPCSTR lpApplicationName,** **LPSTR lpCommandLine,** **LPSECURITY_ATTRIBUTES lpProcessAttr,**	Ստեղծում է պրոցես, որը գործարկում է lpApplicationName անունով կիրառությունը՝ տալով վերջինիս

LPSECURITY_ATTRIBUTES lpThreadAttr, BOOL bInheritHandles, DWORD dwCreationFlags, LPVOID lpEnvironment, LPCSTR lpCurrentDirectory, LPSTARTUPINFO lpStartupInfo, LPPROCESS_INFORMATION lpProcessInfo);	lpCommandLine հրամայական տողը: Ստեղծված պրոցեսի և վերջինիս առաջնային հոսքի նկարագրիչներն ու նույնակիչները գրվում են lpProcessInfo արգումենտում:
void ExitProcess(UINT uExitCode);	Ավարտում է կանչող պրոցեսը uExitCode ելքի կոդով:
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);	Ավարտում է hProcess նկարագրիչի հետ կապված պրոցեսը uExitCode ելքի կոդով:
HANDLE OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessId);	Վերադարձնում է dwProcessId նույնարկիչով պրոցեսի նկարագրիչը՝ տալով վերջինիս dwDesiredAccess հասանելիություն և bInheritHandle-ով որոշվող ժառանգելիություն:
void GetCurrentProcessHandle(PHANDLE hp);	Գրում է կանչող պրոցեսի նկարագրիչը hp արգումենտում:
DWORD GetCurrentProcessId();	Վերադարձնում է կանչող պրոցեսի նույնարկիչը:
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode);	lpExitCode արգումենտում գրում է hProcess նկարագրիչի հետ կապված պրոցեսի ելքի կոդը:

	Եթե պրոցեսը դեռ չի ավարտվել, ապա lpExitCode-ում գրվում է STILL_ACTIVE արժեքը:

Աղյուսակ 13: CreateProcess, ExitProcess, TerminateProcess, OpenProcess, GetCurrentProcessHandle, GetCurrentProcessId և GetExitCodeProcess ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Պրոցեսի ավարտը: Windows օպերացիոն համակարգում պրոցեսը կարող է ավարտվել հետևյալ 4 եղանակներից որևէ մեկով.

1. Պրոցեսի առաջնային հոսքն ավարտում է իր աշխատանքը (return 0; հրամանով): Այս եղանակը լավագույնն է, քանի որ պրոցեսի ավարտը տեղի է ունենում ոչ թե անսպասելիորեն, այլ նախապես որոշված քայլերի հաջորդականության արդյունքում:
2. Պրոցեսի որևէ հոսքի կողմից կանչվում է ExitProcess-ը (Աղ. 13): Այս դեպքում պրոցեսի մյուս հոսքերն ավարտվում են իրենց համար անսպասելիորեն:
3. Որևէ այլ պրոցեսի կողմից կանչվում է TerminateProcess-ը (Աղ. 13): Այս դեպքում պրոցեսն ավարտվում է անսպասելիորեն, ինչը կարող է բերել աշխատանքի թերի կատարման, ռեսուրսների վատ կառավարման և այլն:
4. Պրոցեսում վթարային իրավիճակ է տեղի ունենում (օրինակ՝ կատարվում է 0-ի վրա բաժանում, անհասանելի հիշողության դիմում, առաջանում է չմշակված բացառություն և այլն):

Սպասում: Դիտարկենք, թե պրոցեսի աշխատանքը ժամանակավորապես դադարեցնելու (արգելափակելու) ինչ միջոցներ են առաջարկվում «Windows» օպերացիոն համակարգի կողմից: Նախ կարևոր է նշել, որ վերջինս պրոցեսորային ժամանակի բաշխումը կատարում է ոչ թե պրոցեսների, այլ հոսքերի միջև: Հետևաբար ա-

ռաջարկվող միջոցները նույնպես վերաբերում են հոսքերի արգելափակմանը: Սակայն եթե պրոցեսն ունի միայն առաջնային հոսք, ապա վերջինիս կողմից արգելափակող ֆունկցիայի կանչը կբերի պրոցեսի արգելափակմանը:

Աղյուսակ 14-ի ֆունկցիաներից Sleep-ը արգելափակում է կանչող հոսքն առանց որևէ պայմանի, իսկ WaitForSingleObject-ը և WaitForMultipleObjects-ը տալիս են որևէ միջուկի օբյեկտի կողմից ազդանշանին սպասելու հնարավորություն: Դիտարկենք WaitForSingleObject-ն ավելի մանրամասնորեն: Վերջինս ստանում է ընդամենը 2 արգումենտ՝ hHandle միջուկի օբյեկտի նկարագրիչը և dwMilliseconds մվ-երի քանակը: Ֆունկցիան dwMilliseconds մվ սպասում է hHandle նկարագրիչի հետ կապված միջուկի օբյեկտի կողմից ազդանշանին: Ընդ որում, ազդանշանի իմաստը տարբեր տիպի միջուկի օբյեկտների համար տարբեր է: Օրինակ՝ «պրոցես» և «հոսք» տիպի միջուկի օբյեկտներն ուղարկում են իրենց ավարտի, իսկ մյուտեքսներն ու պատահարները (տե՛ս 4.2)՝ իրենց «ազատվելու» ազդանշանը: Ֆունկցիան վերադարձնում է թվային արժեք, որը կարող է լինել.

- WAIT_FAILED, եթե ֆունկցիայի աշխատանքում սխալ է տեղի ունեցել: Օրինակ՝ hHandle-ն անվավեր նկարագրիչ է կամ չունի SYNCHRONIZE հասանելիությունը (տե՛ս «Microsoft»-ի փաստաթղթեր):
- WAIT_TIMEOUT, եթե dwMilliseconds մվ-ի ընթացքում համապատասխան միջուկի օբյեկտի կողմից սպասվող ազդանշանը չի եկել:
- WAIT_OBJECT_0, եթե dwMilliseconds մվ-ի ընթացքում համապատասխան միջուկի օբյեկտի կողմից սպասվող ազդանշանը եկել է:
- WAIT_ABANDONED-ը կարող է հանդիպել միայն այն դեպքում, երբ hHandle-ը մյուտեքսի նկարագրիչ է: Այս դեպքը կքննարկվի ավելի ուշ:

Ֆունկցիա	Նկարագիր
void Sleep(DWORD dwMilliseconds);	Արգելափակում է կանչող հոսքը dwMilliseconds մվ-ով:
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);	Կանչող հոսքը dwMilliseconds մվ սպասում է hHandle նկարագրիչի հետ կապված միջուկի օբյեկտի կողմից ազդանշանին:
DWORD WaitForMultipleObjects(DWORD nCount, const HANDLE *lpHandles, BOOL bWaitAll, DWORD dwMilliseconds);	dwMilliseconds մվ սպասում է lpHandles նկարագրիչների nCount չափի զանգվածի տարրերի հետ կապված միջուկի օբյեկտների կողմից ազդանշաններին: bWaitAll-ով որոշվում է՝ սպասել պետք է բոլոր օբյեկտների ազդանշաններին, թե՞ գոնե մեկի ազդանշանին:

Աղյուսակ 14: Sleep, WaitForSingleObject և WaitForMultipleObjects ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 8-ում ցուցադրված է “notepad.exe” կիրառության գործարկումը: Ծնողական պրոցեսը «քնում» է 2000 մվ-ով, այնուհետև՝ TerminateProcess համակարգային կանչի միջոցով հարկադրաբար ընդհատում որդիական պրոցեսի աշխատանքը: Այս և հաջորդ ծրագրերում հաշվի է առած այն փաստը, որ պրոցեսի ավարտի ժամանակ փակվում են բոլոր բացված նկարագրիչները. սխալի հետևանքով պրոցեսն ավարտելիս նկարագրիչների փակման համար լրացուցիչ հրամաններ գրված չեն:

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>
```

```
void error_text_output();

int _tmain(){
 STARTUPINFO sInfo{ sizeof(sInfo) };
 PROCESS_INFORMATION pInfo{};

 TCHAR commandLine[] = _T("notepad.exe");
 if (!CreateProcess(NULL,
                    commandLine,
                    NULL,
                    NULL,
                    FALSE,
                    0,
                    NULL,
                    NULL,
                    &sInfo,
                    &pInfo)) {
 error_text_output();
 return 1;
 }

 Sleep(2000);

 if (!TerminateProcess(pInfo.hProcess, 0)) {
 error_text_output();
 return 1;
 }
 CloseHandle(pInfo.hProcess);
 CloseHandle(pInfo.hThread);

 return 0;
}
```

Ծրագիր 8: notepad.exe ծրագրի գործարկումն ու ընդհատումը:

Ծրագիր 9-ում ցուցադրված է “notepad.exe” կիրառության գործարկումը: Ծնողական պրոցեսը «քնում» է 5000 մվ-ով, այնուհետև

GetExitCodeProcess-ի կանչի միջոցով որոշում՝ արդյոք որդիական պրոցեսն ավարտել է աշխատանքը, թե ոչ, և դուրս բերում համապատասխան հաղորդագրություն կոնսոլում։

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

int _tmain() {
 STARTUPINFO sInfo{ sizeof(sInfo) };
 PROCESS_INFORMATION pInfo{};

 TCHAR commandLine[] = _T("notepad.exe");
 if (!CreateProcess(NULL,
                    commandLine,
                    NULL,
                    NULL,
                    FALSE,
                    0,
                    NULL,
                    NULL,
                    &sInfo,
                    &pInfo)) {
 error_text_output();
 return 1;
 }

 Sleep(5000);

 DWORD exit_code{};
 if (!GetExitCodeProcess(pInfo.hProcess, &exit_code)) {
 error_text_output();
 return 1;
 }
 if (exit_code == STILL_ACTIVE) {
```

```
 _tprintf(_T("notepad.exe process is still active\n"));
 }
 else {
 _tprintf(_T("notepad.exe process exited with code: %d\n"), exit_code);
 }
 CloseHandle(pInfo.hProcess);
 CloseHandle(pInfo.hThread);

 return 0;
}
```

Ծրագիր 9: notepad.exe ծրագրի գործարկումն ու ավարտի կոդի դուրսբերումը:

Ծրագիր 10-ում ցուցադրված է “notepad.exe” կիրառության գործարկումը: Մնողական պրոցեսը WaitForSingleObject-ի միջոցով սպասում է որդիական պրոցեսի ավարտին 5000 մվ, այնուհետև՝ դուրս բերում համապատասխան հաղորդագրություն կոնսոլում:

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

int _tmain() {
 STARTUPINFO sInfo{ sizeof(sInfo) };
 PROCESS_INFORMATION pInfo{};

 TCHAR commandLine[] = _T("notepad.exe");
 if (!CreateProcess(NULL,
                    commandLine,
                    NULL,
                    NULL,
                    FALSE,
                    0,
                    NULL,
                    NULL,
```

```
                                &sInfo,
                                &pInfo)) {
error_text_output();
return 1;
}

DWORD r = WaitForSingleObject(pInfo.hProcess, 5000);
switch (r) {
case WAIT_OBJECT_0:
_tprintf(_T("notepad.exe process exited\n"));
break;
case WAIT_TIMEOUT:
_tprintf(_T("notepad.exe process is still active\n"));
break;
case WAIT_FAILED:

error_text_output();
return 1;
}
CloseHandle(pInfo.hProcess);
CloseHandle(pInfo.hThread);

return 0;
}
```

Ծրագիր 10։ notepad.exe ծրագրի գործարկումն ու սպասումը։

Ինֆորմացիայի փոխանակումը պրոցեսների միջև։ Իրական խնդիրներ լուծելիս հաճախ որևէ P1 պրոցեսից որևէ P2 պրոցեսին տարատեսակ ինֆորմացիա փոխանցելու կարիք է լինում։ Այս իրավիճակը հայտնի է՝ որպես միջպրոցեսային համագործակցության առաջին խնդիր։ Քանի որ պրոցեսների հասցեական տարածությունները իրարից առանձնացված են, և ոչ մի պրոցես մյուսի հասցեական տարածությանը հասանելիություն չունի, ապա այդ խնդրի լուծման համար օպերացիոն համակարգի կողմից հատուկ միջոցներ են տրվում։

P1 պրոցեսից P2-ին ինֆորմացիայի փոխանցումը կազմակերպելու համար նախ քննարկենք այն պարզագույն դեպքը, երբ P1-ը ծնողական պրոցես է P2-ի համար: Այս պարագայում առկա են ինֆորմացիան P1-ից P2-ին փոխանցելու հետևյալ ակնհայտ եղանակները.

- Հրամայական տողի օգտագործում:
- Միջավայրի փոփոխականների արժեքների օգտագործում: CreateProcess-ի lpEnvironment արգումենտում NULL արժեքը փոխանցելիս որդիական պրոցեսը ծնողականից ժառանգում է միջավայրի փոփոխականների բլոկը: Այսինքն՝ ստանում է վերջինիս պատճենն իր հասցեական տարածության մեջ: Հետևաբար CreateProcess-ի կանչից առաջ ծնողական պրոցեսը կարող է ստեղծել նոր միջավայրի փոփոխականներ և դրանց արժեքներում գրել փոխանցման ենթակա ինֆորմացիան:
- Նկարագրիչների աղյուսակի ժառանգման օգտագործում: CreateProcess-ի bInheritHandles արգումենտում TRUE արժեքը փոխանցելիս որդիական պրոցեսը ծնողականից ժառանգում է նկարագրիչների աղյուսակը: Այսինքն՝ ստանում է վերջինիս պատճենն իր հասցեական տարածության մեջ: Սակայն ժառանգվում են ոչ թե բոլոր նկարագրիչները, այլ միայն ժառանգելիները: Նկարագրիչը ժառանգելի է, եթե դրա ստեղծման ժամանակ SECURITY_ATTRIBUTES տիպի արգումենտում NULL-ի փոխարեն փոխանցվել է այնպիսի արժեք, որի bInheritHandle դաշտը TRUE է: Հետևաբար կարելի է ծնողական պրոցեսում ստեղծել ժառանգելի նկարագրիչներ, ժառանգել նկարագրիչների աղյուսակը և որդիական պրոցեսում ունենալ արդեն բացված նկարագրիչները:

Ծրագիր 11-ում օգտագործվում է հրամայական տողով որդիական պրոցեսին ինֆորմացիայի փոխանցման մեխանիզմը: Խնդիրը հետևյալն է. գրել 2 ծրագիր՝ A և B: A ծրագիրը գործարկում է B-ն և 5

վրկ հետո ավարտում իր աշխատանքը: B-ն ստուգում է՝ արդյոք A ծրագիրը տրված t ժամանակահատվածում ավարտել է իր աշխատանքը, թե ոչ, և դուրս բերում համապատասխան հաղորդագրություն: Ակնհայտ է, որ A-ի գործարկման արդյունքում ստեղծված պրոցեսը ծնողական է B-ի գործարկման արդյունքում ստեղծված պրոցեսի համար: Օգտագործում է GetExitCodeProcess կամ WaitForSingleObject-երից որևէ մեկը, որպեսզի B-ն կարողանա կատարել ստուգումը: Երկուսին էլ անհրաժեշտ է փոխանցել A-ի նկարագրիչը, որն ունենալու համար B պրոցեսին հարկավոր է փոխանցել A-ի նույնարկիչը: Ծրագիր 11-ում նույնարկիչը փոխանցված է որպես հրամայական տողի արգումենտ, իսկ ստուգման համար օգտագործված է GetExitCodeProcess-ը: Առաջարկվում է խնդիրը լուծել նաև միջավայրի փոփոխականի, այնուհետև՝ նկարագրիչների աղյուսակի ժառանգման օգտագործմամբ (գրել նույնարկիչը որևէ ֆայլում, այդ ֆայլի նկարագրիչը ժառանգել որդիական պրոցեսին):

```
// Code for application A
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

int _tmain() {
 DWORD id = GetCurrentProcessId();
 const int size = 10;
 TCHAR id_as_string[size];
 _itot_s(id, id_as_string, 10);

 STARTUPINFO sInfo{ sizeof(sInfo) };
 PROCESS_INFORMATION pInfo{};

 TCHAR commandLine[MAX_PATH + size + 1];
 _tcscpy_s(commandLine, _T("B.exe ")); // or the full path of executable file
 _tcscat_s(commandLine, id_as_string);
```

```
 if (!CreateProcess(NULL,
                                commandLine,
                                NULL,
                                NULL,
                                FALSE,
                                0,
                                NULL,
                                NULL,
                                &sInfo,
                                &pInfo)) {
 error_text_output();
 return 1;
 }

 Sleep(5000);

 CloseHandle(pInfo.hProcess);
 CloseHandle(pInfo.hThread);

 return 0;
}

// Code for application B
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

int _tmain(int argc, TCHAR* argv[]) {
          if (argc >= 2) {
 const int t = 3000; // or 7000

           DWORD parent_id = _ttoi(argv[1]);
 HANDLE parent_handle =
OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION,
                                                        FALSE,
                                                        parent_id);
```

```
if (NULL == parent_handle) {
error_text_output();
return 1;
}

Sleep(t);

DWORD exit_code{};
if (!GetExitCodeProcess(parent_handle, &exit_code)) {
error_text_output();
return 2;
}
if (exit_code == STILL_ACTIVE) {
_tprintf(_T("Parent process is still active\n"));
}
else {
_tprintf(_T("Parent process exited with code: %d\n"), exit_code);
}
CloseHandle(parent_handle);
        }

        return 0;
}
```

Ծրագիր 11: Ծնողական պրոցեսից որդիականին ինֆորմացիայի փոխանցումը հրամայական տողի միջոցով:

Ծրագիր 12-ում կատարվում է $C_N^K = {}^{N!}/_{K! * (N-K)!}$ զուգորդության հաշվարկի զուգահեռացում պրոցեսների միջոցով: Դիցուք, տրված են N և K բնական թվերը: Պահանջվում է հաշվել C_N^K-ը: Հաշվարկների զուգահեռացման նպատակով ստեղծվում է 3 պրոցես, որոնցից յուրաքանչյուրը հաշվում է $N!$-ը, $K!$-ը և $(N-K)!$-ը առանձին-առանձին: Հաշվարկների արդյունքները որդիական պրոցեսները գրում են ֆայլերում: Ծնողական պրոցեսն սպասում է որդիական պրոցեսների աշխատանքի ավարտին, ֆայլերից կարդում ստացված արդյունքներն ու դուրս բերում C_N^K-ի արժեքը: Ֆայլերի անունները

ծնողական պրոցեսից որդիականներին են փոխանցվում հրամայական տողով, իսկ թիվը, որի համար պետք է կատարել հաշվարկը, միջավայրի փոփոխականի արժեքով: Առաջարկվում է խնդիրը լուծել նաև նկարագրիչների աղյուսակի ժառանգման օգտագործմամբ:

```
// Code for parent process
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

const int BUFFER_SIZE = 10;
const TCHAR* result_file_names[] = { _T("..\\file1.txt"),
                                     _T("..\\file2.txt"),
                                     _T("..\\file3.txt") };

HANDLE create_child_process(int number, int process_number) {
        TCHAR number_as_string[BUFFER_SIZE];
        _itot_s(number, number_as_string, 10);
        if (!SetEnvironmentVariable(number_as_string, number_as_string)) {
                error_text_output();
                return NULL;
        }

        STARTUPINFO sInfo{ sizeof(sInfo) };
        PROCESS_INFORMATION pInfo{};

        TCHAR commandLine[MAX_PATH + 2 * BUFFER_SIZE + 2];
        _tcscpy_s(commandLine, _T("Factorial.exe "));
        _tcscat_s(commandLine, number_as_string);
        _tcscat_s(commandLine, _T(" "));
        _tcscat_s(commandLine, result_file_names[process_number]);
        if (!CreateProcess(NULL,
                           commandLine,
                           NULL,
                           NULL,
```

```
                                FALSE,
                                0,
                                NULL,
                                NULL,
                                &sInfo,
                                &pInfo)) {
                        error_text_output();
                        return NULL;
                }

                CloseHandle(pInfo.hThread);
                return pInfo.hProcess;
}

int _tmain() {
                const int N = 6;
                const int K = 4;
                const int NK = N - K;

                const int child_process_count = 3;
                int numbers[child_process_count] = { N, K, NK };
                HANDLE child_processes[child_process_count]{};
                for (int i = 0; i < child_process_count; ++i) {
                        if ((child_processes[i] =
                         create_child_process(numbers[i], i)) == NULL) {
                                return 1;
                        }
                }

                if (WAIT_FAILED == WaitForMultipleObjects(child_process_count,
                                                        child_processes,
                                                                TRUE,
                                                                INFINITE)) {
                        error_text_output();
                        return 1;
                }

                for (HANDLE h : child_processes) {
```

```
            CloseHandle(h);
        }

        HANDLE result_files[child_process_count]{};
        for (int i = 0; i < child_process_count; ++i) {
            result_files[i] = CreateFile(result_file_names[i],
                                         GENERIC_READ,
                                         0,
                                         NULL,
                                         OPEN_EXISTING,
                                         0,
                                         NULL);
            if (INVALID_HANDLE_VALUE == result_files[i]) {
                error_text_output();
                return 1;
            }
        }

        int result_numbers[child_process_count]{};
        for (int i = 0; i < child_process_count; ++i) {
            TCHAR buffer[BUFFER_SIZE]{};
            DWORD r{};
            if (!ReadFile(result_files[i],
                          buffer,
                          BUFFER_SIZE * sizeof(TCHAR),
                          &r,
                          NULL) || r == 0) {
                error_text_output();
                return 1;
            }
            result_numbers[i] = _ttoi(buffer);
            CloseHandle(result_files[i]);
        }

        int CNK = result_numbers[0] / result_numbers[1] /
result_numbers[2];
        _tprintf(_T("CNK for N = %d, K = %d is : %d\n"), N, K, CNK);
        return 0;
```

```
}

// Code for child processes
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

const int BUFFER_SIZE = 10;

int factorial(int number) {
	int result{ 1 };
	for (int i = 2; i <= number; ++i) {
		result *= i;
	}
	return result;
}

int _tmain(int argc, TCHAR* argv[]) {
	if (argc >= 3) {
		TCHAR buffer[BUFFER_SIZE];
		if (0 == GetEnvironmentVariable(argv[1], buffer,
BUFFER_SIZE)) {
			error_text_output();
			return 1;
		}

		int result = factorial(_ttoi(buffer));
		_itot_s(result, buffer, 10);

		HANDLE h = CreateFile(argv[2],
					GENERIC_WRITE,
					0,
					NULL,
					CREATE_ALWAYS,
					FILE_ATTRIBUTE_NORMAL,
```

```
                                        NULL);
                if (INVALID_HANDLE_VALUE == h) {
                        error_text_output();
                        return 1;
                }
                DWORD w{};
                if (!WriteFile(h,
                                buffer,
                                _tcslen(buffer) * sizeof(TCHAR),
                                &w,
                                NULL) || w == 0) {
                        error_text_output();
                        return 1;
                }
                CloseHandle(h);
        }

        return 0;
}
```

Ծրագիր 12: C_N^K -ի հաշվարկի զուգահեռացումը պրոցեսների օգտագործմամբ:

Պրոցեսից պրոցես ինֆորմացիայի փոխանցման համար հաճախ օգտագործվում են նաև ֆայլեր և կիսվող հիշողություններ (shared memory):

3.3. Աշխատանքը հոսքերի հետ

«Windows» օպերացիոն համարկարգում կամայական պրոցես, բացի իր առաջնային հոսքից, կարող է ունենալ այլ հոսքեր ևս, որոնք կարող են ապահովել զուգահեռ աշխատանք պրոցեսի շրջանակներում: Հոսքը որոշվում է հետևյալ 2 բաղադրիչներով.

- «Հոսք» տիպի միջուկի օբյեկտ, որի միջոցով օպերացիոն համակարգը ղեկավարում է հոսքը: Այս միջուկի օբյեկտն օժտված է թվային նույնարկիչով (իդենտիֆիկատոր), որը

միարժեքորեն որոշում է տվյալ հոսքը՝ օպերացիոն համակարգում աշխատող բոլոր հոսքերի բազմության մեջ:

- Հոսքի հրամանների հաջորդականությունը պարունակող մուտքային ֆունկցիա և պրոցեսի հասցեական տարածության մեջ հատկացված ստեկ (Նկ. 3):

Հոսքի մուտքային ֆունկցիան պետք է ունենա հետևյալ հայտարարությունը.

DWORD WINAPI ThreadProc(LPVOID lpParameter):

Այսինքն՝ մուտքային ֆունկցիան վերադարձնում է թվային արժեք, որը դրա ելքի կոդն է, և որպես lpParameter արգումենտ ստանում է ցուցիչ հիշողության բլոկի վրա՝ առանց տիպային սահմանափակման:

Նկատենք, որ նույն պրոցեսին պատկանող բոլոր հոսքերն աշխատում են այդ պրոցեսի հասցեական տարածության մեջ: Այսինքն՝ բոլորի համար ընդհանուր են գլոբալ տիրույթը, կույտը, նկարագրիչների աղյուսակը և այլն (Նկ. 3):

Հոսքի ստեղծումն ու ավարտը: «Windows» օպերացիոն համակարգում նոր հոսք կարելի է ստեղծել CreateThread համակարգային կանչի միջոցով (Աղ. 16), որը բարեհաջող ավարտի դեպքում վերադարձնում է ստեղծված հոսքի նկարագրիչը և lpThreadId-ում գրում դրա նույնարկիչը:

Հոսքը կարող է ավարտվել հետևյալ 4 եղանակներից որևէ մեկով.

1. Հոսքն ավարտում է իր աշխատանքը ExitThread-ի կանչով (կամ որ նույնն է՝ return հրամանով) (Աղ. 16): Այս եղանակը լավագույնն է, քանի որ հոսքի ավարտը տեղի է ունենում ոչ թե անսպասելիորեն, այլ նախապես որոշված քայլերի հաջորդականության արդյունքում:
2. Հոսքը պարունակող պրոցեսի որևէ այլ հոսք կանչում է ExitProcess-ը (Աղ. 13): Այս դեպքում ավարտվում է պրոցեսը, հետևաբար նաև վերջինիս բոլոր հոսքերը:

3. Որևէ այլ հոսք կանչում է TerminateThread-ը (Աղ. 16)։ Այս դեպքում հոսքն ավարտվում է անսպասելիորեն, ինչը կարող է բերել աշխատանքի թերի կատարման, ռեսուրսների վատ կառավարման և այլն։
4. Հոսքը պարունակող պրոցեսում վթարային իրավիճակ է տեղի ունենում (օրինակ՝ կատարվում է 0-ի վրա բաժանում, անհասանելի հիշողության դիմում, առաջանում է չմշակված բացառություն և այլն)։

Ֆունկցիա	Նկարագիր
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttr, SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);	Կանչող պրոցեսում ստեղծում է նոր հոսք, որին հատկացվում է dwStackSize չափի ստեկ։ Հոսքի մուտքային ֆունկցիայի ցուցիչն lpStartAddr-ն է, իսկ փոխանցվող արգումենտը՝ lpParameter-ը։ Հոսքն ստեղծվում և սկսում է աշխատել dwCreationFlags-ի արժեքին համապատասխան։ Ֆունկցիան վերադարձնում է ստեղծված հոսքի նկարագրիչը, իսկ lpThreadId-ում՝ գրում դրա նույնարկիչը։
void ExitThread(DWORD dwExitCode);	Ավարտում է կանչող հոսքը uExitCode ելքի կոդով։
BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode);	Ավարտում է hThread նկարագրիչի հետ կապված հոսքը uExitCode ելքի կոդով։
HANDLE GetCurrentThread();	Վերադարձնում է կանչող հոսքի նկարագրիչը։

DWORD GetCurrentThreadId();	Վերադարձնում է կանչող հոսքի նույնարկիչը:

Աղյուսակ 16: CreateThread, ExitThread, TerminateThread, OpenProcess, GetCurrentThread և GetCurrentThreadId ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 13-ում կատարվում է $C_N^K = {}^{N!}/_{K! * (N-K)!}$ զուգորդության հաշվարկի զուգահեռացում հոսքերի միջոցով:

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

void error_text_output();

struct Info {
	int m_number;
	int m_result;

	Info(int n)
		: m_number(n)
		, m_result(1)
	{}

	void factorial() {
		for (int i = 2; i <= m_number; ++i) {
			m_result *= i;
		}
	}
};

DWORD WINAPI entry_function(void* parameter) {
	Info* information = static_cast<Info*>(parameter);
	information->factorial();
	return 0;
}
```

```
int _tmain() {
	const int N = 6;
	const int K = 4;
	const int NK = N - K;

	Info* information[] = { new Info(N), new Info(K), new Info(NK) };

	const int thread_count = 3;
	HANDLE thread_handles[thread_count]{};
	DWORD thread_ids[thread_count]{};

	for (int i = 0; i < thread_count; ++i) {
		thread_handles[i] = CreateThread(NULL,
							0,
							entry_function,
							information[i],
							0,
							&thread_ids[i]);
		if (NULL == thread_handles[i]) {
			error_text_output();
			return 1;
		}
	}

	if (WAIT_FAILED == WaitForMultipleObjects(thread_count,

							thread_handles,
								TRUE,
								INFINITE)) {
		error_text_output();
		return 1;
	}

	int CNK = information[0]->m_result /
		 information[1]->m_result /
		 information[2]->m_result;
	_tprintf(_T("CNK for N = %d, K = %d is : %d\n"), N, K, CNK);
```

```
        for (int i = 0; i < thread_count; ++i) {
                CloseHandle(thread_handles[i]);
        }

        return 0;
}
```

Ծրագիր 12: C_N^K -ի հաշվարկի զուգահեռացումը հոսքերի օգտագործմամբ:

Խնդիրներ

1. Գրել ծրագիր, որը գործարկում է notepad.exe-ն և 2 վայրկյանից ավարտում իր աշխատանքը:

2. Գրել ծրագիր, որը գործարկում է notepad.exe-ն փոքրացված (minimized) վիճակում և 2 վայրկյանից ավարտում իր աշատանքը: *Հուշում՝ ուսումնասիրել STARTUPINFO կառուցվածքը և օգտագործել այն CreateProcess ֆունկցիայի կանչի ժամանակ:*

3. Գրել 2 ծրագիր, որում A և B: A ծրագիրը գործարկում է B-ն և 5 վայրկյանից ավարտում իր աշխատանքը: B-ն ստուգում է՝ արդյոք A-ն տրված t ժամանակահատվածում ավարտել է իր աշխատանքը, թե ոչ, և դուրս բերում համապատասխան հաղորդագրություն: Խնդիրը լուծել WaitForSingleObject ֆունկցիայի միջոցով:

4. Գրել ծրագիր, որն իրականացնում է մատրիցի բազմապատկումը վեկտորով՝ զուգահեռացված եղանակով: Խնդիրը լուծել հոսքերի օգտագործմամբ:

5. Գրել ծրագիր, որը տրված թվային միջակայքը տրոհում է n միջակայքերի: n հոսքերի զուգահեռ աշխատանքի միջոցով հաշվել պարզ թվերի քանակը յուրաքանչյուր միջակայքում: Դուրս բերել սկզբնական միջակայքում եղած պարզ թվերի քանակը: Խնդիրը լուծել նաև պրոցեսների օգտագործմամբ:

6. Գրել ծրագիր, որն օգտագործողից մուտքում սպասում է հրամանի անվանում և այն կատարելու համար ծախսվող ժամանակի t վրկ արժեք: Ստացված հրամանը տրվում է կատարման՝ դրա համար ստեղծելով առանձին հոսք: Եթե հրամանը կատարվում է

տրված t ժամանակից ավելի երկար, ապա այն հանվում է կատարումից:

7. (*) Գրել ծրագիր, որը զուգահեռացված եղանակով որոնում է տրված տողն ընթացիկ կատալոգի բոլոր ֆայլերում և դուրս բերում այն ֆայլերի անունները, որոնցում տրված տողը հայտնաբերվել է: Խնդիրը լուծել և՛ հոսքերի, և՛ պրոցեսների օգտագործմամբ:

8. (*) Գրել դաս, որը վերասահմանում է operator new-ն և operator delete-ն այնպես, որ այդ դասի օբյեկտների համար հիշողություն է հատկացվում առանձին կույտից (ստեղծել նոր կույտ): Մեծ քանակով օբյեկտներ ստեղծելով ստուգել՝ արդյոք այդ դասի օբյեկտի ստեղծումն ավելի արագ է, քան համանման այն դասինը, որի համար վերասահմանված չեն վերը նշված օպերատորները: *Հուշում՝ ինքնուրույն ուսումնասիրել կույտի հետ աշխատանքի համար նախատեսված ֆունկցիաները:*

ԳԼՈՒԽ 4
ՍԻՆՔՐՈՆԱՑՈՒՄ

Բազմահոսքային միջավայրում («Windows» օպերացիոն համակարգն այդպիսին է) ծրագրավորելիս պետք է միշտ հաշվի առնել, որ հոսքերն ասինքրոն են աշխատում. նախօրոք անհնար է կանխատեսել, թե որ հոսքը ժամանակի որ պահին ինչ հրաման է կատարում: Սակայն բազմաթիվ խնդիրներում հարկավոր է լինում հոսքերի ասինքրոն պահելաձևը (գուցե մասամբ) սինքրոնացնել: Այսպիսի կարիք առաջանում է 2 իրավիճակում.

- Հոսքերն ընդհանուր ռեսուրս են օգտագործում, հետևաբար այն «փչացնելու»՝ ոչ վավեր վիճակի բերելու պոտենցիալ վտանգ կա (մրցակցային իրավիճակ):
- Հոսքերի աշխատանքի տրամաբանությունը որոշակի հերթականություն է ենթադրում: Օրինակ՝ քանի դեռ t1 հոսքը չի սկզբնարժևորել ինչ-որ տվյալ, t2 և t3 հոսքերը չպետք է սկսեն աշխատել այդ տվյալի հետ:

Հոսքերի սինքրոնացման համար «Windows» օպերացիոն համակարգը տրամադրում է բազմաթիվ գործիքներ: Այս գլուխը նվիրված է դրանց ուսումնասիրությանը: Ընդ որում, դիտարկված են և՛ օգտագործողի (տե՛ս 4.1), և՛ միջուկի (տե՛ս 4.2) մակարդակում սինքրոնացման միջոցները:

4.1. Սինքրոնացումն օգտագործողի մակարդակում

Օգտագործողի մակարդակում սինքրոնացումը ենթադրում է այնպիսի մեթոդների (ֆունկցիաների, օբյեկտների և այլն) օգտագործում, որոնք միջուկի օբյեկտներ չեն: Այսպիսի մեթոդների առավելությունը դրանց արագությունն է, քանի որ չեն օգտագործվում համակարգային կանչեր, տեղի չի ունենում անցում միջուկի ռեժիմի և հակառակը: Սակայն օգտագործողի մակարդակում կարելի է սինքրոնացնել միայն նույն պրոցեսին պատկանող հոսքերի աշխատանքը,

քանի որ վերջիններս կիսում են միմյանց հետ հասցեական տարածության մեծ մասը:

Interlocked ֆունկցիաների ընտանիքը: Այս ընտանիքի ֆունկցիաների միջոցով հնարավոր է ատոմար (չընդհատվող) հասանելիություն ապահովել որևէ տարրական տիպի փոփոխականի նկատմամբ: Interlocked ֆունկցիաները շատ են: Աղյուսակ 19-ում բերված են դրանցից մի քանիսը: Դրանք նախատեսված են՝ ամբողջաթիվ արժեքը մեկով ավելացնելու (InterlockedIncrement), պակասեցնելու (InterlockedDecrement), թիվ գումարելու (InterlockedAdd) և որևէ այլ արժեքով փոխելու (InterlockedExchenge) համար: Interlocked ընտանիքի բոլոր ֆունկցիաներին, ինչպես նաև դրանց օգտագործման օրինակներին կարելի է ծանոթանալ «Microsoft»-ի պաշտոնական փաստաթղթերի էջում:

Ֆունկցիա	Նկարագիր
LONG InterlockedIncrement(LONG volatile* Addend);	Addend ցուցչի ցույց տված արժեքն ատոմար ձևով ավելացնում է մեկով և վերադարձնում ստացված արժեքը:
LONG InterlockedDecrement(LONG volatile* Addend);	Addend ցուցչի ցույց տված արժեքն ատոմար ձևով պակասեցնում է մեկով և վերադարձնում ստացված արժեքը:
LONG InterlockedAdd(LONG volatile* Addend, LONG Value);	Addend ցուցչի ցույց տված արժեքին ատոմար ձևով գումարում է Value և վերադարձնում ստացված արժեքը:
LONG InterlockedExchange(LONG volatile* Target, LONG Value);	Target ցուցչի ցույց տված արժեքն ատոմար ձևով փոխարինում է Value-ով և վերադարձնում ստացված արժեքը:

Աղյուսակ 19: InterlockedIncrement, InterlockedDecrement, InterlockedAdd և InterlockedExchange ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

CRITICAL_SECTION կառուցվածքը: Ծրագրի այն հատվածը, որը մրցակցային իրավիճակ է առաջացնում, անվանենք կրիտիկական: Մրցակցային իրավիճակում սինքրոնացում կատարելու համար սովորաբար հարկավոր է ապահովել, որ իրավիճակին մասնակցող բոլոր հոսքերն իրենց կրիտիկական հատվածում աշխատեն փոխադարձ բացառման ձևով: Այսինքն, եթե հոսքերից որևէ մեկն արդեն սկսել է կատարել իր կրիտիկական հատվածի հրամանները, ապա մյուս հոսքերից և ոչ մեկը չպետք է կարողանա մտնել իր կրիտիկական հատվածը: Այսպիսի սինքրոնացումը լիովին լուծում է մրցակցային իրավիճակում ընդհանուր ռեսուրսը «փչացնելու» խնդիրը: Օգտագործողի մակարդակում այդպիսի սինքրոնացում կարելի է ապահովել՝ CRITICAL_SECTION կառուցվածքի օբյեկտ օգտագործելով:

CRITICAL_SECTION կառուցվածքի օբյեկտն օգտագործելուց առաջ հարկավոր է այն սկզբնարժևորել InitializeCriticalSection, իսկ օգտագործելուց հետո՝ ջնջել՝ DeleteCriticalSection ֆունկցիան կանչելով (Աղ. 20): Այնուհետև հարկավոր է սինքրոնացվող հոսքերում առանձնացնել կրիտիկական հատվածները և դրանցից առաջ կատարել EnterCriticalSection, իսկ հետո LeaveCriticalSection ֆունկցիաաների կանչերը: Երաշխավորվում է, որ եթե հոսքերից որևէ մեկը կանչել է EnterCriticalSection-ը և սկսել իր կրիտիկական հատվածի կատարումը, ապա մյուս բոլոր հոսքերը կարգելափակվեն իրենց կրիտիկական հատվածին նախորդող EnterCriticalSection կանչի պահին: Այդ հոսքերը կկարողանան դուրս գալ արգելափակումից, երբ առաջինը կավարտի իր կրիտիկական հատվածի կատարումը և կկանչի LeaveCriticalSection ֆունկցիան:

Ֆունկցիա	**Նկարագիր**
void InitializeCriticalSection(**LPCRITICAL_SECTION** **lpCriticalSection** **);**	Սկզբնարժևորում է lpCriticalSection կրիտիկական հատվածը:

void DeleteCriticalSection(**LPCRITICAL_SECTION** **lpCriticalSection** **);**	Ջնջում է lpCriticalSection կրիտիկական հատվածը:
void EnterCriticalSection(**LPCRITICAL_SECTION** **lpCriticalSection** **);**	Փորձում է զբաղեցնել lpCriticalSection կրիտիկական հատվածը: Եթե վերջինս արդեն զբաղված է, ապա սպասում է:
BOOL TryEnterCriticalSection(**LPCRITICAL_SECTION** **lpCriticalSection** **);**	Փորձում է զբաղեցնել lpCriticalSection կրիտիկական հատվածը: Եթե վերջինս արդեն զբաղված է, ապա վերադարձնում է FALSE:
void LeaveCriticalSection(**LPCRITICAL_SECTION** **lpCriticalSection** **);**	Ազատում է lpCriticalSection կրիտիկական հատվածը:

Աղյուսակ 20: InitializeCriticalSection, DeleteCriticalSection, EnterCriticalSection, TryEnterCriticalSection և LeaveCriticalSection ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 14-ում պրոցեսի առաջնային հոսքն ստեղծում է 2 հոսք, որոնցից առաջինը գլոբալ փոփոխականի արժեքն ավելացնում, իսկ երկրորդը պակասեցնում է պատահական թվով: Գլոբալ փոփոխականի նկատմամբ առաջացած մրցակցային իրավիճակում կատարվում է հոսքերի սինքրոնացում CRITICAL_SECTION կառուցվածքի օբյեկտի միջոցով: Առաջարկվում է նույն խնդիրը լուծել՝ Interlocked ընտանիքի ֆունկցիաներ օգտագործելով:

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>
```

```
CRITICAL_SECTION cs;
int variable = 0;

void error_text_output();

DWORD WINAPI first_thread(void*) {
	srand(0);
	int count = rand() % 100;
	for (int i = 0; i < count; ++i) {
		EnterCriticalSection(&cs);
		variable += rand();
		_tprintf(_T("Global variable's value on %d-st iteration is
%d\n"),
			i,
			variable);
		LeaveCriticalSection(&cs);
	}

	return 0;
}

DWORD WINAPI second_thread(void*) {
	srand(1);
	int count = rand() % 100;
	for (int i = 0; i < count; ++i) {
		EnterCriticalSection(&cs);
		variable -= rand();
		_tprintf(_T("Global variable's value %d-st iteration is
%d\n"),
			i,
			variable);
		LeaveCriticalSection(&cs);
	}

	return 0;
}

int _tmain() {
```

```
	InitializeCriticalSection(&cs);

	HANDLE thread1_handle = CreateThread(NULL, 0, first_thread,
NULL, 0, NULL);
	if (NULL == thread1_handle) {
		error_text_output();
		return 1;
	}

	HANDLE thread2_handle = CreateThread(NULL, 0, second_thread,
NULL, 0, NULL);
	if (NULL == thread2_handle) {
		error_text_output();
		return 1;
	}

	HANDLE handles[2] = { thread1_handle, thread2_handle };
	if (WAIT_FAILED == WaitForMultipleObjects(2, handles, TRUE,
INFINITE)) {
		error_text_output();
		return 1;
	}

	CloseHandle(thread1_handle);
	CloseHandle(thread2_handle);
	DeleteCriticalSection(&cs);

	_tprintf(_T("Global variable's final value is %d\n"), variable);

	return 0;
}
```

Ծրագիր 14: Հոսքերի սինքրոնացումը CRITICAL_SECTION կառուցվածի օբյեկտի միջոցով:

4.2. Սինքրոնացումը միջուկի մակարդակում

Տարբեր պրոցեսներին պատկանող հոսքերի սինքրոնացումը հնարավոր է կատարել միայն միջուկի մակարդակում՝ օգտագործելով միջուկի օբյեկտներ: Այդպիսի օբյեկտներ են մյուտեքսները, սեմաֆորները և պատահարները: Դիտարկենք դրանցից յուրաքանչյուրն առանձին:

Մյուտեքս: Անգլերեն mutex բառը mutual exclusion բառակապակցության կրճատումն է, որը թարգմանվում է որպես «փոխադարձ բացառում»: Մյուտեքսներն օգտագործվում են մրցակցային իրավիճակներում հոսքերի սինքրոնացման համար (ինչպես CRITICAL_SECTION-ը): Մյուտեքսը միջուկի օբյեկտ է, որն ունի հետևյալ հատուկ դաշտերը.

- անուն, որը միարժեքորեն որոշում է տվյալ մյուտեքսը գոյություն ունեցող բոլոր մյուտեքսների բազմության մեջ,
- պատկանելիություն, որը որոշում է, թե մյուտեքսը որ հոսքին է պատկանում: Մյուտեքսը «Windows» օպերացիոն համակարգում միակ միջուկի օբյեկտն է, որը կարող է պատկանել որևէ հոսքի: Այդ պատկանելիությունը որոշելու համար պահվում է այն հոսքի նույնարկիչը, որին պատկանում է մյուտեքսը: Եթե այդ դաշտում գրված է 0, ապա մյուտեքսն ազատ է. չի պատկանում ոչ մի հոսքի: Այլ հոսքեր կարող են սպասել մյուտեքսին Wait ֆունկցիաներից որևէ մեկով՝ ունենալով մյուտեսքի նկարագրիչը: Այդ սպասումն արգելափակում կառաջացնի, եթե մյուտեքսն զբաղված է. պատկանում է որևէ հոսքի, և կավարտվի հաջողությամբ, եթե մյուտեքսն ազատ է. չի պատկանում ոչ մի հոսքի:

Մյուտեքս ստեղծելու համար պետք է կանչել CreateMutex, ազատելու համար՝ ReleaseMutex ֆունկցիաները: OpenMutex ֆունկցիան օգտագործվում է՝ արդեն գոյություն ունեցող մյուտեքսի համար նկարագրիչ ստեղծելու նպատակով (Աղ. 21):

Ֆունկցիա	Նկարագիր
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpAttr, BOOL bInitialOwner, LPCSTR lpName);	Ստեղծում է lpName անունով, lpAttr պաշտպանական ատրիբուտներով մյուտեքս միջուկի օբյեկտ և վերադարձնում վերջինիս նկարագրիչը: Մյուտեքսի պատկանելիությունը որոշվում է bInitialOwner արժեքով. այն պատկանում է ֆունկցիան կանչող հոսքին, եթե այդ արժեքը TRUE է, և ազատ է հակառակ դեպքում:
BOOL ReleaseMutex(HANDLE hMutex);	Ազատում է hMutex նկարագրիչի հետ կապված մյուտեքսը:
HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName);	Վերադարձնում է lpName անունով մյուտեքսի նկարագրիչը՝ տալով վերջինիս dwDesiredAccess հասանելիություն և bInheritHandle-ով որոշվող ժառանգելիություն:

Աղյուսակ 21: CreateMutex, ReleaseMutex և OpenMutex ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Windows-ական մյուտեքսների հետ աշխատանքում հնարավոր է, որ առաջանա հետևյալ իրավիճակը: Դիցուք, m մյուտեքսը պատկանում է t1 հոսքին, իսկ t2 հոսքն սպասում է m-ին WaitForSingleObject ֆունկցիայի միջոցով: Եթե t1 հոսքը որևէ պատճառով «մոռանա» ազատել (ReleaseMutex կանչել) m մյուտեքսն ու ավարտի իր աշխատանքը, ապա m-ը կհայտնվի անվավեր վիճակում. այն կպատկանի իրականում այլևս գոյություն չունեցող, ավարտված մի հոսքի: Հարց է առաջանում՝ ի՞նչ կպատահի t2 հոսքի հետ: Այդպի-

սի իրավիճակում հայտնված մյուտեքսը փաստացի հանդիսանում է «անտեր», և դրան սպասող Wait ֆունկցիան դուրս կգա սպասումից WAIT_ABANDONED արժեքով։ Այսինքն՝ t2 հոսքը չի հայտնվի անժամկետ արգելափակման մեջ։ Սակայն այս դեպքում t2-ը պետք է հասկանա, որ մյուտեքսը գտնվում է անվավեր վիճակում, և մեծ է հավանականությունը, որ վերջինիս միջոցով պաշտպանվող ռեսուրսը նույնպես հայտնվել է անվավեր վիճակում։

Սեմաֆոր։ Սեմաֆորները նույնպես օգտագործվում են մրցակցային իրավիճակներում հոսքերի սինքրոնացման համար։ Սակայն այնպիսի դեպքերում, երբ ռեսուրսի նկատմամբ հարկավոր է ապահովել ոչ թե մեկ, այլ մի քանի հոսքերի ատոմար աշխատանք։ Սեմաֆորը միջուկի օբյեկտ է, որն ունի հետևյալ հատուկ դաշտերը.

- Անուն, որը միարժեքորեն որոշում է տվյալ սեմաֆորը՝ գոյություն ունեցող բոլոր սեմաֆորների բազմության մեջ։
- Առավելագույն քանակ, որը որոշում է, թե պաշտպանվող ռեսուրսի հետ առավելագույնը քանի հոսք կարող է աշխատել միաժամանակ։ Այս արժեքը պետք է դրական լինի։
- Ընթացիկ քանակ, որը որոշում է, թե քանի հոսք դեռ կարող է սկսել աշխատել պաշտպանվող ռեսուրսի հետ։ Այս արժեքը պետք է դրական լինի ու չգերազանցի առավելագույն քանակի արժեքը։ Այլ հոսքեր կարող են սպասել սեմաֆորին Wait ֆունկցիաներից որևէ մեկով՝ ունենալով սեմաֆորի նկարագրիչը։ Այդ սպասումն արգելափում կառաջացնի, եթե սեմաֆորն զբաղված է. ընթացիկ քանակը 0 է, և կավարտվի հաջողությամբ, եթե սեմաֆորն ազատ է. ընթացիկ քանակը մեծ է 0-ից։ Հաջողությամբ ավարտված սպասումը նվազեցնում է ընթացիկ քանակը 1-ով։

Սեմաֆոր ստեղծելու համար պետք է կանչել CreateSemaphore, ընթացիկ քանակն աճեցնելու, այսինքն՝ սեմաֆորն ազատելու համար՝ ReleaseSemaphore ֆունկցիաները։ OpenSemaphore ֆունկցիան օգտագործվում է՝ արդեն գոյություն ունեցող սեմաֆորի համար նկարագրիչ ստեղծելու նպատակով (Աղ. 22)։

Ֆունկցիա	Նկարագիր
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpAttrs, LONG lInitialCount, LONG lMaximumCount, LPCSTR lpName);	Ստեղծում է lpName անունով, lpAttr պաշտպանական ատրիբուտներով սեմաֆոր միջուկի օբյեկտ և վերադարձնում վերջինիս նկարագրիչը: Սեմաֆորի առավելագույն քանակը որոշվում է lMaximumCount, իսկ ընթացիկ քանակը՝ lInitialCount արժեքներով:
BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCount);	Փոխում է hSemaphore նկարագրիչով որոշվող սեմաֆորի ընթացիկ քանակը lReleaseCount-ով՝ գրելով lpPreviousCount-ում ընթացիկ նախորդ քանակի արժեքը:
HANDLE OpenSemaphore(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName);	Վերադարձնում է lpName անունով սեմաֆորի նկարագրիչը՝ տալով վերջինիս dwDesiredAccess հասանելիություն և bInheritHandle-ով որոշվող ժառանգելիություն:

Աղյուսակ 22: CreateSemaphore, ReleaseSemaphore և OpenSemaphore ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 15-ում լուծված է արտադրող-սպառողի խնդիրը (հայտնի է նաև որպես սահմանափակ զանգվածի խնդիր)՝ օգտագործելով 2 սեմաֆոր և 1 մյուտեքս: Խնդրի դասական ձևակերպումը հետևյալն է: Ունենք $n \geq 1$ «արտադրող» և $m \geq 1$ «սպառող» հոսքեր, որոնք աշխատում են սահմանափակ և անփոփոխ չափ ունեցող զանգվածի հետ: «Արտադրող» հոսքերից յուրաքանչյուրն անընդհատ գեներացնում է (արտադրում է) ինֆորմացիա և գրում այն զանգվածում:

«Սպառող» հոսքերից յուրաքանչյուրն անընդհատ կարդում է ինֆորմացիան զանգվածից, հանում այն և օգտագործում (սպառում)։ Խնդիրը հետևյալն է. ինչպես կազմակերպել «արտադրող» և «սպառող» հոսքերի աշխատանքները, որպեսզի չլինեն ինֆորմացիայի կորուստ, կրկնօրինակում, անվավերացում և այլն, ինչպես նաև «արտադրող» հոսքերը չփորձեն գրել լցված զանգվածում, իսկ «սպառող» հոսքերը՝ կարդալ դատարկ զանգվածից։

```
#include <Windows.h>
#include <stdio.h>
#include <tchar.h>

const int MAX_ITEMS_FOR_PRODUCER = 50;
const int MAX_ITEMS_FOR_CONSUMER = 50;
const int PRODUCER_QTY = 5;
const int CONSUMER_QTY = 5;
const int BUFFER_SIZE = 80;

HANDLE empty, full, mutex;
int buffer[BUFFER_SIZE];
int in = 0, out = 0;

void error_text_output();

DWORD WINAPI producer(void* p) {
 const int producer_number = *static_cast<int*>(p);
 int item{};
 for (int i = 0; i < MAX_ITEMS_FOR_PRODUCER; ++i) {
 item = rand();
 WaitForSingleObject(empty, INFINITE);
 WaitForSingleObject(mutex, INFINITE);
 buffer[in] = item;
 _tprintf(_T("Producer %d: Insert Item %d at %d\n"),
                        producer_number,
                        buffer[in],
                        in);
```

```
 in = (in + 1) % BUFFER_SIZE;
 ReleaseMutex(mutex);
 ReleaseSemaphore(full, 1, NULL);
 }
 delete p;
 return 0;
}

DWORD WINAPI consumer(void* p) {
 const int consumer_number = *static_cast<int*>(p);
 for (int i = 0; i < MAX_ITEMS_FOR_CONSUMER; ++i) {
 WaitForSingleObject(full, INFINITE);
 WaitForSingleObject(mutex, INFINITE);
 int item = buffer[out];
 _tprintf(_T("Consumer %d: Remove Item %d from %d\n"),
                        consumer_number,
                        item,
                        out);
 out = (out + 1) % BUFFER_SIZE;
 ReleaseMutex(mutex);
 ReleaseSemaphore(empty, 1, NULL);
 }
 delete p;
 return 0;
}

int _tmain() {
 empty = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
 full = CreateSemaphore(NULL, 0, BUFFER_SIZE, NULL);
 mutex = CreateMutex(NULL, FALSE, NULL);
 if (NULL == empty || NULL == full || NULL == mutex) {
 error_text_output();
 return 1;
 }

 HANDLE producers[PRODUCER_QTY]{};
 for (int i = 0; i < PRODUCER_QTY; ++i) {
 producers[i] = CreateThread(NULL, 0, producer, new int(i), 0, NULL);
```

```
if (NULL == producers[i]) {
error_text_output();
return 1;
}
}

HANDLE consumers[CONSUMER_QTY]{};
for (int i = 0; i < CONSUMER_QTY; ++i) {
consumers[i] = CreateThread(NULL, 0, consumer, new int(i), 0, NULL);
if (NULL == consumers[i]) {
error_text_output();
return 1;
}
}

if (WAIT_FAILED == WaitForMultipleObjects(PRODUCER_QTY,
                                          producers,
                                          TRUE,
                                          INFINITE)) {
error_text_output();
return 1;
}
if (WAIT_FAILED == WaitForMultipleObjects(CONSUMER_QTY,
                                          consumers,
                                          TRUE,
                                          INFINITE)) {
error_text_output();
return 1;
}

for (int i = 0; i < PRODUCER_QTY; ++i) {
CloseHandle(producers[i]);
}
for (int i = 0; i < CONSUMER_QTY; ++i) {
CloseHandle(consumers[i]);
}
CloseHandle(mutex);
CloseHandle(empty);
```

```
CloseHandle(full);

return 0;
}
```

Ծրագիր 15։ Մյուտեքսերի և սեմաֆորների օգտագործումն արտադրող-սպառողի խնդրի լուծման մեջ։

Պատահար (event)։ Պատահարներն օգտագործվում են հոսքերի աշխատանքի որոշակի հերթականության ապահովման համար։ Պատահարը միջուկի օբյեկտ է, որն ունի հետևյալ հատուկ դաշտերը.

- Անուն, որը միարժեքորեն որոշում է տվյալ պատահարը՝ գոյություն ունեցող բոլոր պատահարների բազմության մեջ։
- Վիճակ։ Պատահարը կարող է լինել կամ ազատ, կամ զբաղված։ Այլ հոսքեր կարող են սպասել պատահարին Wait ֆունկցիաներից որևէ մեկով՝ ունենալով պատահարի նկարագրիչը։ Այդ սպասումն արգելափակում կառաջացնի, եթե պատահարն զբաղված է, և կավարտվի հաջողությամբ, եթե պատահարն ազատ է։
- Տեսակ։ Պատահարը կարող է լինել կա՛մ ավտոմատ (auto reset), կա՛մ ոչ ավտոմատ (manual reset)։ Եթե հաջողությամբ է ավարտվել սպասումն ավտոմատ պատահարին, ապա վերջինս ավտոմատ ձևով դառնում է զբաղված։ Հակառակ դեպքում պատահարը պետք է դարձնել զբաղված ResetEvent ֆունկցիայի կանչով։

Պատահար ստեղծելու համար պետք է կանչել CreateEvent, «ազատ» վիճակի բերելու համար՝ SetEvent, իսկ «զբաղված» վիճակի բերելու համար՝ ResetEvent ֆունկցիաները։ OpenEvent ֆունկցիան օգտագործվում է՝ արդեն գոյություն ունեցող պատահարի համար նկարագրիչ ստեղծելու նպատակով (Աղ. 23)։

Ֆունկցիա	**Նկարագիր**
HANDLE CreateEvent(	Ստեղծում է lpName անունով,

LPSECURITY_ATTRIBUTES lpAttrs, **BOOL bManualReset,** **BOOL bInitialState,** **LPCSTR lpName** **);**	lpAttr պաշտպանական ատրիբուտներով պատահար միջուկի օբյեկտ և վերադարձնում վերջինիս նկարագրիչը: Պատահարի տեսակը որոշվում է bManualReset, իսկ սկզբնական վիճակը՝ bInitialState արժեքներով:
BOOL SetEvent(**HANDLE hEvent** **);**	Ազատում է hEvent նկարագրիչի հետ կապված պատահարը:
BOOL ResetEvent(**HANDLE hEvent** **);**	Զբաղեցնում է hEvent նկարագրիչի հետ կապված պատահարը:
HANDLE OpenEvent(**DWORD dwDesiredAccess,** **BOOL bInheritHandle,** **LPCTSTR lpName** **);**	Վերադարձնում է lpName անունով պատահարի նկարագրիչը՝ տալով վերջինիս dwDesiredAccess հասանելիություն և bInheritHandle-ով որոշվող ժառանգելիություն:

Աղյուսակ 23: CreateEvent, SetEvent, ResetEvent և OpenEvent ֆունկցիաների հայտարարությունը և հակիրճ նկարագիրը:

Ծրագիր 16-ում լուծված է հետևյալ խնդիրը: Գրել ծրագիր, որն ստեղծում է 2 հոսք: Առաջին հոսքը գլոբալ զանգվածի առաջին կեսում գրում է 1-եր, ապա պատահարի միջոցով ազդարարում այդ աշխատանքի ավարտը և սպասում 2-րդ հոսքին: Վերջինս զանգվածի 2-րդ կեսում գրում է 2-ներ, ապա պատահարի միջոցով ազդարարում իր աշխատանքի ավարտը: Դրանից հետո առաջին հոսքը զանգվածի պարունակությունը դուրս է բերում կոնսոլում:

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

```
#include <tchar.h>

const int ARRAY_SIZE = 1000;
int ARRAY[ARRAY_SIZE];
HANDLE event1_handle, event2_handle;

void error_text_output();

DWORD WINAPI first_thread(void*) {
	for (int i = 0; i < ARRAY_SIZE / 2; ++i) {
		ARRAY[i] = 1;
	}

	SetEvent(event1_handle);

	WaitForSingleObject(event2_handle, INFINITE);

	for (int i = 0; i < ARRAY_SIZE; ++i) {
		_tprintf(_T("%d "), ARRAY[i]);
	}
	_tprintf(_T("\n"));

	return 0;
}

DWORD WINAPI second_thread(void*) {
	WaitForSingleObject(event1_handle, INFINITE);

	for (int i = ARRAY_SIZE / 2; i < ARRAY_SIZE; ++i) {
		ARRAY[i] = 2;
	}

	SetEvent(event2_handle);
	return 0;
}

int _tmain() {
	event1_handle = CreateEvent(NULL, FALSE, FALSE, NULL);
```

```
	if (NULL == event1_handle) {
		error_text_output();
		return 1;
	}

	event2_handle = CreateEvent(NULL, FALSE, FALSE, NULL);
	if (NULL == event2_handle) {
		error_text_output();
		return 1;
	}

	HANDLE thread1_handle = CreateThread(NULL, 0, first_thread,
NULL, 0, NULL);
	if(NULL == thread1_handle) {
		error_text_output();
		return 1;
	}
	HANDLE thread2_handle = CreateThread(NULL, 0, second_thread,
NULL, 0, NULL);
	if (NULL == thread2_handle) {
		error_text_output();
		return 1;
	}

	HANDLE handles[2] = { thread1_handle, thread2_handle };
	if (WAIT_FAILED == WaitForMultipleObjects(2, handles, TRUE,
INFINITE)) {
		error_text_output();
		return 1;
	}

	CloseHandle(event1_handle);
	CloseHandle(event2_handle);
	CloseHandle(thread1_handle);
	CloseHandle(thread2_handle);

	return 0;
}
```

Ծրագիր 16: Պատահարների օգտագործումը:

Խնդիրներ

1. Գրել ծրագիր, որն ստեղծում է 2 հոսք: Հոսքերից մեկը լցնում է գլոբալ զանգվածը պատահական թվերով: Մյուս հոսքը ժամանակ առ ժամանակ (որևէ t ժամանակ) կարդում է հերթական թիվը զանգվածից և դուրս բերում կոնսոլում: Սինքրոնացնել հոսքերի աշխատանքն այնպես, որ եթե մի հոսքն աշխատում է զանգվածի հետ, ապա մյուս հոսքն այդ պահին չկարողանա դիմել դրան: Խնդիրը լուծել և՛ կրիտիկական հատվածի, և՛ մյուտեքսի օգտագործմամբ:

2. Գրել ծրագիր, որն ստեղծում է 3 հոսք: Առաջին հոսքը գլոբալ զանգվածը լցնում է պատահական թվերով: Երկրորդ և երրորդ հոսքերը կարդում են զանգվածի պարունակությունը և դուրս բերում կոնսոլում: Պատահարների միջոցով սինքրոնացնել հոսքերի աշխատանքն այնպես, որ երկրորդ և երրորդ հոսքերը չսկսեն աշխատել, քանի դեռ առաջինը չի վերջացրել իր աշխատանքը:

3. Գրել ծրագիր, որն ստեղծում է 2 հոսք: Առաջին հոսքը գլոբալ զանգվածի առաջին N տարրերին վերագրում է 1-ից N թվերը և պատահարի միջոցով ազդարարում իր աշխատանքի ավարտը: Երկրորդ հոսքը M վայրկյան սպասում է առաջինի աշխատանքի ավարտին, և եթե այդ ընթացքում առաջինն ավարտել է իր աշխատանքը, զանգվածի առաջին N տարրերը դուրս է բերում կոնսոլում, հակառակ դեպքում արտածում է համապատասխան հաղորդագրություն:

4. Գրել ծրագիր, որն ստեղծում է 2 հոսք: Առաջին հոսքը փորձում է գլոբալ զանգվածը լցնել 1-երով, երկրորդ հոսքը՝ 2-ներով: Վերջում առաջնային հոսքն արտածում է զանգվածի պարունակությունը: Պատահարների միջոցով ապահովել, որ զանգվածում ստացվի հետևյալ տեսքի հաջորդականություն՝ 1 2 1 2 1 2 1 2...

5. Գրել ծրագիր, որն ստեղծում է 2 հոսք: Առաջին հոսքը փորձում է գոլոբալ զանգվածը լցնել 1-երով, երկրորդ հոսքը՝ 2-ներով: Վերջում առաջնային հոսքն արտածում է զանգվածի պարունակությունը: Պատահարների միջոցով ապահովել, որ զանգվածում ստացվի հետևյալ տեսքի հաջորդականություն՝ 1 1 2 2 1 1 2 2...

6. Գրել ծրագիր, որն ստեղծում է 2 հոսք։ Հոսքերից մեկում կազմակերպել N երկարությամբ ցիկլ, որի յուրաքանչյուր քայլին վայրկյանը մեկ անգամ պատահար ազդարարվի։ Մյուս հոսքը, հետևելով պատահարին, 1-ից սկսած, հերթական թիվը դուրս է բերում կոնսոլում։

7. (*) *«Ապահով» գծային ցուցակի իրականացումը։* Ստեղծել thread_safe_list դաս, որն իրականացնում է «միակապ գծային ցուցակ» տվյալների կառույցն այնպես, որ տարր որոնելու, ավելացնելու և հեռացնելու գործողություններն «ապահով» են զուգահեռացված աշխատանքի համար։ Դիցուք, այդ դասի օբյեկտի հետ կարող են աշխատել 3 տիպի հոսքեր՝ որոնող, ավելացնող և հեռացնող։ Կանոնները հետևյալն են.

- Որոնող հոսքերի աշխատանքի ժամանակ բացառվում է ավելացնող և հեռացնող հոսքերի աշխատանքը, բայց թույլատրվում է այլ որոնողների աշխատանքը։
- Ավելացնող հոսքերի աշխատանքի ժամանակ բացառվում է այլ հոսքերի աշխատանքը։
- Հեռացնող հոսքերի աշխատանքի ժամանակ բացառվում է այլ հոսքերի աշխատանքը։

8. (*) *Մայր թռչնի և նրա ճուտիկների կյանքի մոդելավորումը։* Դիցուք, ունենք մայր թռչուն և n ճուտիկներ։ Ճուտիկները սնվում են նույն ամանից, որում մայր թռչունն ուտելիք է ավելացնում։ Սկզբում ամանի մեջ կա F բաժին ուտելիք։ Ամեն ճուտիկ ուտում է մեկ բաժին ուտելիք, քնում է որոշ ժամանակ, հետո նորից ուտում և այդպես շարունակ։ Երբ ուտելիքը վերջանում է, վերջինն ուտող ճուտիկը կանչում է մայր թռչնին։ Մայրը լցնում է ամանը F բաժին ուտելիքով և նորից սպասում դրա դատարկվելուն։ Ապահովել, որ ճուտիկներն ուտելիս իրար «չխանգարեն», ինչպես նաև «չխանգարեն» մորը, երբ վերջինս ուտելիք է լցնում ամանի մեջ։

9. (*) *Արտադրող-սպառողի խնդիրը։* Լուծել արտադրող-սպառողի դասական խնդիրն օբյեկտային կողմնորոշված նախագծմամբ (OOD)։ Ընդհանրացնել խնդիրը m արտադրողի և n սպառողի հա-

մար: Իրականացնել կոնկրետ ապրանքի տիպ որևէ իրական ոլորտի համար:

10. (*) *Քառասյին շարժման մոդելավորումը:* Դիցուք, կոորդինատային հարթության վրա ունենք որևէ ուղղանկյունաձև տիրույթ (թույլատրված տիրույթ) և տրված քանակի գնդիկների սկզբնական կոորդինատներն այդ տիրույթում: Գնդիկների կոորդինատները ժամանակի ընթացքում փոփոխվում են պատահական սկզբունքով: Ընդ որում, եթե գնդիկը դուրս է գալիս թույլատրված տիրույթից, ապա անհետանում է: Իրականացնել գնդիկների կոորդինատների փոփոխության զուգահեռացումը և՛ պրոցեսների, և՛ հոսքերի օգտագործմամբ:

11. (*) *Պատերազմի մոդելավորումը:* Դիցուք, ունենք 2 հակառակորդ զորք: Յուրաքանչյուր զորքի զինվորների քանակն աճում է պատահական սկզբունքով, և ամեն զորք սպանում է պատահական քանակի զինվոր հակառակորդ զորքից: Եթե զորքերից մեկում զինվորների քանակը հատում է նախապես որոշված նվազագույնի սահմանը, ապա հայտարարել այդ զորքին պարտված, իսկ մյուսին՝ հաղթած: Իրականացնել զորքերի քանակային փոփոխությունների զուգահեռացումը և՛ պրոցեսների, և՛ հոսքերի օգտագործմամբ:

12. (*) *Վինի Փուհի և մեղուների կյանքի մոդելավորումը:* Դիցուք, ունենք տրված քանակի մեղուներ, որոնք ժամանակի ընթացքում հավասար քանակների մեղր են բերում փեթակ: Վինի Փուհն ինչ-որ հաճախությամբ գողանում է որոշակի քանակի մեղր: Մոդելավորել մեղուների և Վինի Փուհի կյանքն այնպես, որ մեղուները «չխանգարեն» Փուհին ապրել: Իրականացումը կատարել և՛ պրոցեսների, և՛ հոսքերի օգտագործմամբ:

ՕԳՏԱԳՈՐԾՎԱԾ ԳՐԱԿԱՆՈՒԹՅԱՆ ՑԱՆԿ

1. A. S. Tanenbaum, H. Boss, Modern operating systems, 4th edition, Vrije Universiteit, Amsterdam, The Netherlands, Pearson Education, 1133 p., 2015.

2. J. M. Hart, Windows system programming, 4th edition, Westford, US, Pearson Education, 646 p., 2010.

3. J. Richter, Programming applications for Microsoft Windows, 4th edition, Microsoft Press, 342 p., 1999.

4. *docs.microsoft.com*. Microsoft documentation for end users, developers, and IT professionals.

5. J. Richter, Ch. Nasarre, Windows via C/C++, 5th edition, Microsoft Press, 739 p., 2008.

6. W. Stallings, Operating systems: internals and design principles, 9th edition, Pearson Education, 800 p., 2018.

7. A. S. Tanenbaum, A. S. Woodhull, Operating systems design and implementation, 3rd edition, Pearson Education, 1088 p., 2008.

8. W. S. Davis, T. M. Rajkumar, Operating systems: a systematic view, 6th edition, Pearson Education, 688 p., 2005.

ԵՐԵՎԱՆԻ ՊԵՏԱԿԱՆ ՀԱՄԱԼՍԱՐԱՆ

Սվետա Ավետիսյան, Անի Քոչարյան
Մինաս Հովհաննիսյան

«WINDOWS» ՕՊԵՐԱՑԻՈՆ ՀԱՄԱԿԱՐԳԵՐ

(ուսումնամեթոդական ձեռնարկ)

Համակարգչային ձևավորումը՝ Կ. Չալաբյանի
Կազմի ձևավորումը՝ Ա. Պատվականյանի
Հրատ. խմբագրումը՝ Մ. Հովհաննիսյանի

Ստորագրված է տպագրության՝ 25.06.2021:
Չափսը՝ 60x84 $^{1}/_{16}$: Տպ. մամուլը՝ 5,625:
Տպաքանակը՝ 100:

ԵՊՀ հրատարակչություն
ք. Երևան, 0025, Ալեք Մանուկյան 1
www.publishing.am