SO Cheat Sheet
Lab 1 - Introducere

Linux

gcc

```
gcc [-c|-S|-E] [-std=standard]
    [-g] [-pg] [-Olevel]
    [-Wwarn...] [-pedantic]
    [-Idir...] [-Ldir...]
    [-Dmacro[=defn]...] [-Umacro]
    [-o outfile] [@file] infile...
```

Fazele compilării

- **-E**, preprocesare fişier sursă
  - gcc -o hello.i -E hello.c
- **-S**, compilare fişier sursă
  - gcc -o hello.s -S hello.c
- **-c**, asamblare fişier sursă
  - gcc -o hello.o -c hello.c

Alte opţiuni

- **-Wall**, activarea tuturor avertismentelor.
- **-lxlib**, caută biblioteca xlib la editarea de legaturi.
- **-Ldir**, adaugă dir la lista directoarelor căutate pentru -l
- **-Idir**, adaugă dir la începutul listei de căutare pentru fişierele antet.
- **-DMACRO[=val]**, defineşte macro în linia de comanda.
- **-g**, genereare simboluri de debug folosite mai târziu de debugger.

make

```
make [ -f makefile ] [ options ] ... [ targets ]
```

```
target: dependinte
<tab>    comanda
```

Variabile uzuale

- **CC**, defineşte compilatorul folosit
- **CFLAGS**, defineşte flag-urile de compilare
- **LDLIBS**, defineşte bibliotecile folosite la editarea de legături
- **$@**, se expandează la numele target-ului
- **$^**, se expandează la lista de dependinţe
- **$<**, se expandează la prima dependinţă

gdb

```
gdb prog [arguments]
```

Executabilul trebuie compilat cu flag-ul **-g**

Comenzi:

- **b [file:]function**, setează brakepoint la funcţia dată.
- **r arglist**, run - rulează programul cu argumentele date.
- **bt**, backtrace - afişează stiva programului
- **p expr**, print - afişează valoarea lui expr
- **c**, continue - continuă rularea programului după oprirea la un breakpoint
- **n**, next - execută următoare linie de program, trece peste orice apel de funcţie pe acea linie
- **s**, step - execută următoarea linie de program, trece prin orice apel de funcţie de pe acea linie.
- **quit**, ieşire din GDB.

Creare biblioteci

- **statice**
  - creare arhivă
    - **ar rc libX_static.a X1.o X2.o**
  - legare bibliotecă
    - **gcc -Wall main.c -o main -lX_static -L.**
- **dinamice**
  - **creare obiect partajat**
    - **gcc -shared f1.o f2.o -o libX_shared.so**
  - **legare bibliotecă**
    - `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.`
    - **gcc -Wall main.c -o main -lX_shared -L.**

Windows

cl

```
CL [option...] file... [option | file]... [lib...]  [/link link-opt
```

Opţiuni

- **/c**, realizează doar compilare fără editare de legături.
- **/Wall**, activează toate avertismentele
- **/D**, definire macrou în linie de comandă
- **/I<dir>**, adaugă dir la începutul listei de căutare pentru fişierele antet.
- **/LIBPATH<dir>**, indică editorului de legături să caute şi în dir bibliotecile pe care le va folosi programul

Specificare fişiere de ieşire

- **/Fa<file>**, specifică numele fişierului în limbaj de asamblare.
- **/Fo<file>**, specifică numele fişierului obiect.
- **/Fe<file>**, specifică numele fişierului executabil.

nmake

```
NMAKE [option...] [macros...] [targets...]
```

- **/F**, pentru a rula alt fişier make decât cel implicit cu numele Makefile.

Creare biblioteci

- **statice**
  - se foloseşte comanda **lib**
  - **lib /out:intro.lib f1.obj f2.obj**
- **dinamice**
  - precizarea explicită a simbolurilor folosite
    - **__declspec(dllimport)**, folosit pentru a importa o funcţie
    - **__declspec(dllexport)**, folosit pentru a exporta o funcţie
  - creare bibliotecă dinamică şi bibliotecă de import
    - folosind opţiunea **/LD** a compilatorului cl
      - **cl /LD f1.obj f2.obj**
    - folosind comanda link
      - **link /nologo /dll /out:intro.dll /implib:intro.lib f1.obj f2.obj**

2. Operaţii I/O simple

Linux

File descriptor

Întreg ce identifică un fişier în tabela fişierelor deschise de un proces.

File descriptori standard:

- **0** sau **STDIN_FILENO**, standard input
- **1** sau **STDOUT_FILENO**, standard output
- **2** sau **STDERR_FILENO**, standard error

Operaţii pe fişiere

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

`pathname`	calea către fişiere
`flags`	flag-uri de access şi de creare
`mode`	permisiuni la creare
întoarce	file descriptorul creat sau -1 în caz de eroare

```
int close(int fd);
```

`fd`	descriptor fişier
întoarce	0 în caz de succes sau -1 în caz de eroare

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

`fd`	descriptor fişier
`buf`	adresa de început a zonei de memorie pentru citire/scriere
`count`	numărul de octeţi din **buf** solicitaţi pentru citire/scriere
întoarce	numărul de octeţi citiţi/scrişi sau -1 în caz de eroare

```
off_t lseek(int fd, off_t offset, int whence);
```

`fd`	descriptor fişier
`offset`	offset folosit pentru poziţionare
`whence`	poziţia relativă la care se face poziţionarea
întoarce	offset rezultat după poziţionare măsurat relativ la începutul fişierului

```
int unlink(const char *pathname);
```

`pathname`	numele fişierului care va fi şters
întoarce	zer în caz de success, -1 in caz de eroare

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

`oldfd`	vechiul file descriptor
`newfd`	noul file descriptor care va deveni acum o copie a vechiului file descriptor
întoarce	noul file descriptor

Windows

File handle

Identificator pentru un obiect gestionat de kernel-ul Windows.
Device-uri standard:

- **STD_INPUT_HANDLE**, standard input device
- **STD_OUTPUT_HANDLE**, standard output device
- **STD_ERROR_HANDLE**, standard error device

```
BOOL WINAPI SetStdHandle(DWORD nStdHandle, HANDLE hHandle);
```

- **nStdHandle** - device-ul standard pentru care va fi setat handle-ul
- **hHandle** - handle-ul pentru standard device
- *întoarce* - nonzero pentru succes, zero în caz de eroare

```
HANDLE WINAPI GetStdHandle(DWORD nStdHandle);
```

- **nStdHandle** - device-ul standard pentru care va fi obţinut handle-ul
- *întoarce* - handle-ul obţinut sau INVALID_HANDLE_VALUE în caz de eroare

Operaţii pe fişiere

```
HANDLE CreateFile(
    LPCTSTR lpFileName, DWORD dwDesiredAccess,
    DWORD dwShareMode, LPSECURITY_ATTRIBUTES
    lpSecurityAttributes, DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes, HANDLE hTemplateFile );
```

- **lpFileName** - numele fişierului creat sau deschis
- **dwDesiredAccess** - GENERIC_READ, GENERIC_WRITE
- **dwShareMode** - FILE_SHARE_READ, FILE_SHARE_WRITE, FILE_SHARE_DELETE
- **lpSecurityAttributes** - de obicei NULL
- **dwCreationDisposition** - CREATE_ALWAYS, CREATE_NEW, OPEN_ALWAYS, OPEN_EXISTING, TRUNCATE_EXISTING
- **dwFlagsAndAttributes** - FILE_ATTRIBUTE_NORMAL, FILE_ATTRIBUTE_READONLY
- **hTemplateFile** - de obicei NULL
- *întoarce* - handle-ul pentru fişier sau INVALID_HANDLE_VALUE în caz de eroare

```
BOOL CloseHandle(HANDLE hObject);
```

- **hObject** - handle care se doreşte a fi închis
- *întoarce* - nonzero pentru succes, zero în caz de eroare

```
BOOL DeleteFile(LPCTSTR lpFileName);
```

- **lpFileName** - numele fişierului care se doreşte a fi şters
- *întoarce* - nonzero pentru succes sau zero în caz de eroare

```
BOOL ReadFile( HANDLE hFile, LPVOID lpBuffer, DWORD
nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped );
```

- **hFile** - handle către un fişier deschis
- **lpBuffer** - buffer în care se vor reţine octeţii citiţi din fişier
- **nNumberOfBytesToRead** - numărul de octeţi de citit
- **lpNumberOfBytesRead** - numărul de octeţi efectiv citiţi
- **lpOverlapped** - momentan NULL
- *întoarce* - nonzero pentru succes sau zero în caz de eroare

```
BOOL WriteFile( HANDLE hFile, LPCVOID lpBuffer, DWORD
nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten,
LPOVERLAPPED lpOverlapped );
```

- **hFile** - handle către un fişier deschis
- **lpBuffer** - buffer care se va scrie în fişier
- **nNumberOfBytesToWrite** - numărul de octeţi de scris
- **lpNumberOfBytesWritten** - numărul de octeţi efectiv scrişi
- **lpOverlapped** - momentan NULL
- *întoarce* - nonzero pentru succes sau zero în caz de eroare

```
 DWORD SetFilePointer( HANDLE hFile, LONG lDistanceToMove,
PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod );
```

- **hFile** - handle către un fişier deschis
- **lDistanceToMove** - numărul de octeţi cu care se mută cursorul
- **lpDistanceToMoveHigh** - de obicei NULL
- **dwMoveMethod** - poziţia relativă faţă de care se face mutarea
  - FILE_BEGIN, punctul de start este începutul fişierului
  - FILE_CURRENT, punctul de start este valoarea curentă a cursorului
  - FILE_END, punctul de start este valoarea curentă a sfârşitului de fişier
- *întoarce* - noua valoarea a cursorului în cazul în care **lpDistanceToMoveHigh** este NULL, sau INVALID_HANDLE_VALUE în caz de eroare.

3. Procese

Linux

Operații cu procese

```
int system(const char *command);
```

`command` comanda de executat
întoarce 0 în caz de succes sau -1 în caz de eroare

```
pid_t fork(void);
```

întoarce 0 în copil, pid în părinte, -1 în caz de eroare

```
pid_t getpid(void);
```

întoarce PID-ul procesului apelant

```
pid_t getppid(void);
```

întoarce PID-ul părintelui procesului apelant

```
int execl(const char *path, const char *arg, ...);
```

`path` calea către program
`arg` lista variabilă de parametri; numele primului parametru coincide cu cel al programului; ultimul parametru este NULL
întoarce doar în caz de eroare

```
int execvp(const char *file, char *const argv[], ...);
```

`file` numele executabilului; cautat în calea reținută de PATH
`argv` vector parametri; numele primului parametru coincide cu cel al programului; ultimul parametru este NULL
întoarce _doar_ în caz de eroare (altfel nu se întoarce)

```
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status);
```

`pid` pid-ul procesului ce se dorește așteptat, 0 sau negativ pentru comportamente speciale
`status` informația de terminare
`options` diverse opțiuni, în general 0
întoarce 0 în caz de succes sau -1 în caz de eroare

```
void exit(int status);
```

`status` codul de terminare a procesului

Variabile de mediu

```
int main(int argc, char **argv, char **environ);
```

`environ` vector de șiruri de forma VARIABILĂ = VALOARE

```
char* getenv(const char *name);
```

`name` numele variabilei
întoarce valoarea variabilei sau NULL dacă nu există

```
int setenv(const char *name, const char *value, int replace)
```

`name` numele variabilei
`value` valoarea variabilei
`replace` 1 dacă variabila este deja definită și se dorește înlocuirea vechii valori
întoarce 0 în caz de succes sau -1 în caz de eroare

```
char* unsetenv(const char *name);
```

`name` numele variabilei
întoarce 0 în caz de succes sau -1 în caz de eroare

Pipe-uri

```
int pipe(int filedes[2]);
```

`filedes` vector de descriptori ai capetelor pipe-ului cu 0 - citire, 1 - scriere
întoarce 0 în caz de succes sau -1 în caz de eroare

```
int mkfifo(const char *pathname, mode_t mode);
```

`pathname` calea din sistemul de fișiere
`mode` permisiunile
întoarce 0 în caz de succes sau -1 în caz de eroare

Windows

Operații cu procese

```
BOOL CreateProcess( LPCTSTR lpApplicationName, LPTSTR
lpCommandLine, LPSECURITY_ATTRIBUTES lpProcessAttributes,
LPSECURITY_ATTRIBUTES lpThreadAttributes, BOOL
bInheritHandles, DWORD dwCreationFlags, LPVOID
lpEnvironment, LPCTSTR lpCurrentDirectory, LPSTARTUPINFO
lpStartupInfo, LPPROCESS_INFORMATION lpProcessInformation );
```

- **lpApplicationName** - numele modulului de executat
- **lpCommandLine** - linia de comandă de executat
- **lpProcessAttributes** - atributele procesului
- **lpThreadAttributes** - atributele firului principal
- **bInheritHandles** - TRUE dacă se dorește moștenirea descriptorilor în procesele create
- **dwCreationFlags** - stabilește clasa de prioritate a procesului
- **lpEnvironment** - environment block-ul
- **lpCurrentDirectory** - directorul curent
- **lpStartupInfo** - atribute auxiliare, ex: redirectări
- **lpProcessInformation** - parametru out, populat de funcție cu diverse informații
- **întoarce** - nonzero pentru succes, zero în caz de eroare

```
DWORD WaitForSingleObject( HANDLE hHandle, DWORD
dwMilliseconds );
```

- **hHandle** - handle-ul procesului ce se dorește așteptat
- **dwMilliseconds** - numărul maxim de milisecunde de așteptare, de obicei INFINITE
- **întoarce** - WAIT_FAILED în caz de eroare

```
void ExitProcess( UINT uExitCode );
```

- **uExitCode** - codul de iesire al procesului

```
BOOL TerminateProcess( HANDLE hProcess, UINT uExitCode );
```

- **hProcess** - handle-ul procesului ce se dorește terminat
- **uExitCode** - codul de ieșire al procesului
- **întoarce** - nonzero pentru succes, zero în caz de eroare

```
BOOL DuplicateHandle( HANDLE hSourceProcessHandle, HANDLE
hSourceHandle, HANDLE hTargetProcessHandle, LPHANDLE
lpTargetHandle, DWORD dwDesiredAccess, BOOL bInheritHandle,
DWORD dwOptions );
```

- **hSourceProcessHandle** - handle-ul procesului proprietar al descriptorului ce se dorește duplicat
- **hSourceHandle** - descriptorul ce se dorește duplicat
- **hTargetProcessHandle** - handle-ul procesului doritor al handle-ului duplicat
- **lpTargetHandle** - handle-ul duplicat
- **dwDesiredAccess** - drepturile de acces
- **bInheritHandle** - TRUE dacă se dorește ca noul handle să poată fi moștenit mai departe
- **dwOptions** - optiuni suplimentare
- **întoarce** - nonzero pentru succes, zero în caz de eroare

Variabile de mediu

```
LPTCH GetEnvironmentStrings(void);
```

- **întoarce** - un șir cu perechi VARIABILĂ = VALOARE

```
BOOL FreeEnvironmentStrings( LPTSTR lpszEnvironmentBlock );
```

- **lpszEnvironmentBlock** - pointer obținut prin GetEnvironmentStrings
- **întoarce** - nonzero pentru succes, zero în caz de eroare

```
DWORD GetEnvironmentVariable( LPCTSTR lpName, LPTSTR
lpBuffer, DWORD nSize );
```

- **lpName** - numele variabilei
- **lpBuffer** - zona în care funcția va depune valoarea
- **nSize** - dimensiunea zonei de mai sus

- **întoarce** - numărul de caractere ale valorii(dacă zona este suficient de încăpătoare), numărul de caractere necesar(dacă zona nu este suficient de încăpătoare), zero în caz de eroare

```
BOOL SetEnvironmentVariable( LPCTSTR lpName, LPCTSTR
lpValue);
```

- **lpName** - numele variabilei
- **lpBuffer** - noua valoare sau NULL dacă se dorește înlăturarea variabilei
- **întoarce** - nonzero pentru succes, zero în caz de eroare

Pipe-uri

```
BOOL CreatePipe( PHANDLE hReadPipe, PHANDLE hWritePipe,
LPSECURITY_ATTRIBUTES lpPipeAttributes, DWORD nSize );
```

- **hReadPipe** - descriptorul capătului de citire
- **hWritePipe** - descriptorul capătului de scriere
- **lpPipeAttributes** - determină posibilitatea moștenirii
- **nSize** - dimensiunea, în bytes, a buffer-ului intern
- **întoarce** - nonzero pentru succes, zero în caz de eroare

```
HANDLE CreateNamedPipe( LPCTSTR lpName, DWORD dwOpenMode,
DWORD dwPipeMode, DWORD nMaxInstances, DWORD nOutBufferSize,
DWORD nInBufferSize, DWORD nDefaultTimeOut,
LPSECURITY_ATTRIBUTES lpSecurityAttributes );
```

- **lpName** - șir ce desemnează numele pipe-ului
- **dwOpenMode** - stabilește o serie de caracteristici, precum sensul simplu sau dublu de circulație a informației
- **dwPipeMode** - flux de octeti sau mesaj
- **nMaxInstances** - numărul maxim de instanțe
- **nOutBufferSize** - dimensiunea buffer-ului de ieșire
- **nInBufferSize** - dimensiunea buffer-ului de intrare
- **nDefaultTimeOut** - durata implicită de așteptare până ce o instanță a pipe-ului devine disponibilă
- **lpSecurityAttributes** - controlează posibilitatea de moștenire a handle-ului
- **întoarce** - handle-ul capătului dinspre server al pipe-ului, INVALID_HANDLE_VALUE în caz de eroare

```
BOOL ConnectNamedPipe( HANDLE hNamedPipe, LPOVERLAPPED
lpOverlapped );
```

- **hNamedPipe** - handle-ul capătului dinspre server al pipe-ului
- **lpOverlapped** - oferă un mecanism de notificare la apariția unei noi cereri, când se lucrează în mod asincron
- **întoarce** - în mod sincron nonzero pentru succes, zero în caz de eroare; în mod asincron, funcția întoarce zero, iar starea operației este descrisă de GetLastError

SO Cheat Sheet
Semnale

Linux

Trebuie inclus header-ul `signal.h`

Descrierea semnalelor

`char *strsignal(int sig)` – întoarce descrierea textuală a unui semnal

`void psignal(int sig, const char *s)` – afişează descrierea textuală a unui semnal, alături de mesajul dat ca parametru

Măşti de semnale

`int sigemptyset(sigset_t *set)` – elimină toate semnalele din mască

`int sigfillset(sigset_t *set)` – adaugă toate semnalele la mască

`int sigaddset(sigset_t *set, int signo)` – adaugă semnalul precizat la mască

`int sigdelset(sigset_t *set, int signo)` – elimină semnalul precizat din mască

`int sigismember(sigset_t *set, int signo)` – verifică daca semnalul precizat aparţine măştii

Blocarea semnalelor

`int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)` – obţine sau modifică masca de semnale a firului apelant

`how`	unul dintre `SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`
`set`	masca ce conţine noile semnale blocate/deblocate
`oldset`	vechea mască de semnale
întoarce	0 succes, -1 eroare

Tratarea semnalelor

`sighandler_t signal(int signum, sighandler_t handler)` – stabileşte acţiunea efectuată la primirea unui semnal

`signum`	numărul semnalului
`handler`	una din valorile `SIG_IGN`, `SIG_DFL` sau adresa unei funcţii de tratare
întoarce	adresa handler-ului anterior sau `SIG_ERR` în caz de eroare

`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` – stabileşte acţiunea efectuată la primirea unui semnal

`signum`	numărul semnalului
`act`	noua acţiune de executat
`oldact`	vechea acţiune
întoarce	0 succes, -1 eroare

Semnalarea proceselor

`int kill(pid_t pid, int sig)` – trimite un semnal unui proces, fără a garanta recepţia

`pid`	procesul destinaţie
`sig`	semnalul trimis
întoarce	0 succes, -1 eroare

`int sigqueue(pid_t pid, int signo, const union sigval value)` – trimite un semnal unui proces, garantând recepţia

`pid`	procesul destinaţie
`signo`	semnalul trimis
`value`	informaţie suplimentară, ce însoţeşte semnalul, şi care poate fi obţinută din câmpul `siginfo_t->si_value`
întoarce	0 succes, -1 eroare

Aşteptarea semnalelor

`int sigsuspend(const sigset_t *mask)` – înlocuieşte, temporar, masca de semnale, şi se blochează în aşteptarea unui semnal neblocat

`mask`	masca temporară
întoarce	întotdeauna -1

Timer-e

`int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid)` – crearea unui timer

`clockid`	specifică tipul ceasului: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID
`evp`	specifică modul de notificare la expirarea timer-ului
`timerid`	întoarce identificatorul timer-ului
întoarce	0 succes, -1 eroare

`int timer_settime(timer_t timerid, int flags, const struct itimerspec *new_value, struct itimerspec * old_value)` – armarea unui timer

`timerid`	identificator timer
`flags`	poate fi 0 sau TIMER_ABSTIME
`new_value`	noii parametrii ai timer-ului
`old_value`	vechii parametrii
întoarce	0 succes, -1 eroare

`int timer_delete(timer_t timerid)` – ștergerea unui timer

`timerid`	identificator timer
întoarce	0 succes, -1 eroare

Windows

Waitable Timer Objects

`HANDLE WINAPI CreateWaitableTimer(LPSECURITY_ATTRIBUTES lpAttributes, BOOL bManualReset, LPCTSTR lpTimerName)` – creează sau deschide un timer

- **lpAttributes** - permite moştenirea handle-ului timer-ului în procesele copil
- **bManualReset** - dacă este `TRUE`, timer-ul rămâne în starea *signaled* până ce se apelează `SetWaitableTimer`
- **lpTimerName** - numele timer-ului
- **întoarce** - handle-ul timer-ului, `NULL` în caz de eroare

`BOOL WINAPI SetWaitableTimer(HANDLE hTimer, const LARGE_INTEGER *pDueTime, LONG lPeriod, PTIMERAPCROUTINE pfnRoutine, LPVOID lpRoutineArg, BOOL fResume)` – creează sau deschide un timer

- **hTimer** - handle-ul timer-ului
- **pDueTime** - primul interval, după care expiră timer-ul, în multipli de 100 ns
- **lPeriod** - perioada timer-ului, în milisecunde
- **pfnRoutine** - adresa funcţiei executate la expirare (opţional)
- **lpRoutineArg** - parametrul funcţiei de tratare (opţional)
- **fResume** - dacă este `TRUE`, şi timer-ul intră în starea *signaled*, sistemul aflat în starea de conservare a energiei îşi reia activitatea
- **întoarce** - `TRUE` pentru succes

`BOOL WINAPI CancelWaitableTimer(HANDLE hTimer)` – dezactivează un timer

- **hTimer** - handle-ul timer-ului

`DWORD WINAPI WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds)` – permite aşteptarea expirării unui timer

SO Cheat Sheet
5. Gestiunea Memoriei

WINDOWS

`HANDLE HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize)`

- **flOptions** - opțiuni pentru alocarea heapului (poate fi 0 sau HEAP_CREATE_ENABLE_EXECUTE, HEAP_GENERATE_EXCEPTIONS, HEAP_NO_SERIALIZE)
- **dwInitialSize** - dimensiunea inițială de memorie rezervată heapului (în octeți)
- **dwMaximumSize** - dimensiunea maximă în octeți (0 - nu e limitată)
- **întoarce** - handle către noul heap (NULL în caz de eroare)

`BOOL HeapDestroy(HANDLE hHeap)`

- **hHeap** - handle către heap
- **întoarce** - o valoare diferită de 0 în caz de succes (pentru detalii despre eroare GetLastError)

`LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags,SIZE_T dwBytes)`

- **hHeap** - handle către heap
- **dwFlags** - suprascrie valoarea specificată de HeapCreate: HEAP_GENERATE_EXCEPTIONS, HEAP_NO_SERIALIZE, HEAP_ZERO_MEMORY
- **dwBytes** - numărul de octeți
- **întoarce** - pointer către blocul de memorie

`LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags,LPVOID lpMem, SIZE_T dwBytes)`

- **hHeap** - handle către heap
- **dwFlags** - suprascrie valoarea specificată prin flOptions: HEAP_GENERATE_EXCEPTIONS, HEAP_NO_SERIALIZE, HEAP_REALLOC_IN_PLACE_ONLY, HEAP_ZERO_MEMORY
- **lpMem** - pointer către blocul de memorie
- **dwBytes** - noua dimensiune a blocului de memorie specificată în octeți
- **întoarce** - pointer către blocul de memorie

În caz de eroare, atât HeapAlloc, cât și HeapReAlloc, dacă nu s-a specificat HEAP_GENERATE_EXCEPTIONS, va întoarce NULL, altfel va genera una din următoarele excepții STATUS_NO_MEMORY sau STATUS_ACCESS_VIOLATION

`BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem)`

- **hHeap** - handle către heap
- **dwFlags** - suprascrie valoarea specificată de HeapCreate (HEAP_NO_SERIALIZE)
- **lpMem** - pointer către blocul de memorie
- **întoarce** - o valoare diferită de 0 în caz de succes (pentru detalii despre eroare GetLastError)

LINUX

`void *malloc(size_t size)` – alocă memorie neinițializată

`size`	numărul de octeți
întoarce	un pointer către memoria alocată

`void *calloc(size_t nmemb, size_t size)` – alocă memorie inițializată cu zero

`nmemb`	numărul de elemente al vectorului
`size`	dimensiunea în octeți a unui element
întoarce	un pointer către memoria alocată

`void *realloc(void *ptr, size_t size)` – modifică dimensiunea blocului de memorie

`ptr`	pointer către blocul de memorie
`size`	dimensiunea în octeți
întoarce	un pointer către noua zonă de memorie alocată

Atât `malloc`, cât și `calloc` și `realloc` întorc NULL în caz de eroare.

`void free(void *ptr)` – dezalocarea unei zone de memorie

`ptr`	pointer către zona de memorie

Dacă ptr este NULL nu se execută nici o operație.

MTRACE

Trebuie inclusă biblioteca **mcheck.h**

`void mtrace(void)` – activează monitorizarea apelurilor de bibliotecă de lucru cu memoria

`void muntrace(void)` – dezactivează monitorizarea apelurilor de bibliotecă de lucru cu memoria

GDB

- **gdb "file"**
- **quit**
- **help**
- **run** - pornește execuția
- **kill** - oprește programul
- **break "FUNC"** - setează breakpoint la începutul unei funcții
- **break "ADDR"** - setează breakpoint la adresa specificată
- **nexti "NUM"** - execută NUM instrucțiuni
- **continue** - reia execuția
- **backtrace** - afișează toate apelurile de funcții în curs de execuție
- **info reg** - afișează conținutul registrelor
- **disassamble** - afișează codul mașină generat de compilator

VALGRIND

valgrind –tool=memcheck ./executabil

SO Cheat Sheet
Memoria virtuală

Mapare fișiere și memorie POSIX

În POSIX trebuie incluse header-ele **sys/types.h**, **sys/mman.h**.

`void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)` – mapare fișier/memorie în spațiul de adresă al unui proces

`start`	adresa de start pentru mapare, NULL înseamna lipsa unei preferințe; dacă e diferită de NULL, e considerată doar un hint, maparea fiind creată la o adresă apropiată multiplu de dimensiunea unei pagini
`length`	lungimea mapării
`prot`	tipul de acces la zona de memorie: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
`flags`	tipul de mapare: cel puțin una din MAP_PRIVATE/MAP_SHARED, MAP_FIXED, MAP_LOCKED; MAP_ANONYMOUS pentru mapare memorie
`fd`	descriptorul fișierului mapat; ignorat la mapare memorie
`offset`	offset în cadrul fișierului mapat; ignorat la mapare memorie
întoarce	adresa mapării, MAP_FAILED (care reprezintă (void *) -1) în caz de eroare

`int msync(void *start, size_t length, int flags)` – declanșează în mod explicit sincronizarea fișierului cu maparea din memorie

`start, length`	identifică zona de memorie
`flags`	MS_SYNC, MS_ASYNC, MS_INVALIDATE
întoarce	0 în caz de succes, -1 în caz de eroare

`int munmap(void *start, size_t length)` – demapează o zonă din spațiul de adresă al procesului ; poate identifica zone aparținând unor mapari diferite, unele deja demapate

`void *mremap(void *old_address, size_t old_size, size_t new_size, unsigned long flags)` – redimensionare zonă mapată

`old_address, old_size`	identifică vechea zonă mapată
`new_size`	noua dimensiune
`flags`	MREMAP_MAYMOVEE
întoarce	pointer spre noua zonă în caz de succes, MAP_FAILED în caz de eroare

`int mprotect(const void *addr, size_t len, int prot)` – schimbă drepturile de acces ale unei mapări

`addr`	adresa mapării, multiplu de dimensiunea unei pagini
`len`	lungimea zonei considerate; se va aplica pentru toate paginile cu cel puțin un byte în intervalul [addr, addr + len -1]
`prot`	PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
întoarce	0 în caz de succes, -1 în caz de eroare

`int madvise(void *start, size_t length, int advice)` – indicații despre cum va fi folosită o zonă mapată

`addr, length`	identifică zona
`advice`	MADV_NORMAL, MADV_RANDOM, MADV_SEQUENTIAL, MADV_WILLNEED, MADV_DONTNEED
întoarce	0 în caz de succes, -1 în caz de eroare

Blocarea paginării POSIX

`int mlock(const void *addr, size_t len)` – blochează paginarea paginilor incluse în intervalul [addr, addr + len - 1]

`int mlockall(int flags)` – blochează paginarea tuturor paginilor procesului

`flags`	MCL_CURRENT, MCL_FUTURE

`int munlock(const void *addr, size_t len)` – reporni paginarea tuturor paginilor din intervalul [addr, addr + len - 1]

`int munlockall(void)` – reporni paginarea tuturor paginilor procesului

Excepții accesare memorie POSIX

Funcția de tip sigaction va primi ca parametru o structură siginfo_t, având setate:

`si_signo`	SIGSEGV, SIGBUS
`si_code`	caz SIGSEGV: SEGV_MAPPER, SEGV_ACCERR; caz SIGBUS: BUS_ADRALN, BUS_ADRERR, BUS_OBJERR
`si_addr`	adresa care a generat excepția

ElectricFence

Folosit pentru depanare buffer overrun. Pentru a preveni și buffer underrun, definiți variabila de mediu EF_PROTECT_BELOW.

Programul trebuie linkat cu efence: -lefence.

Maparea fișierelor Win32

`HANDLE CreateFileMapping( HANDLE hFile, LPSECURITY_ATTRIBUTES lpAttributes, DWORD flProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR lpName )` – crează un obiect FileMapping, pentru a fi mapat

- **hFile** - handle fișier de mapat
- **lpAttributes** - atribute securitate pentru acces handle întors
- **flProtect** - tipul mapării: PAGE_READONLY, PAGE_READWRITE, PAGE_WRITECOPY
- **dwMaximumSizeHigh, dwMaximumSizeLow** - dimensiunea maximă
- **lpName** - șir identificare, opțional
- **întoarce** - în caz de succes întoarce un handle către un obiect FileMapping, în caz de eșec întoarce NULL

`LPVOID MapViewOfFile( HANDLE hFileMappingObject, DWORD dwDesiredAccess, DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, SIZE_T dwNumberOfBytesToMap )` – creează mapare fișier

- **hFileMappingObject** - obiect de tip FileMapping
- **dwDesiredAccess** - FILE_MAP_READ, FILE_MAP_WRITE, FILE_MAP_COPY
- **dwFileOffsetHigh, dwFileOffsetLow** - offset
- **dwNumberOfBytesToMap** - număr octeți de mapat
- **întoarce** - în caz de succes întoarce un pointer la zona mapată, în caz de eșec întoarce NULL

`BOOL UnmapViewOfFile( LPCVOID lpBaseAddress )` – demapare fișier mapat în memorie

- **lpBaseAddresst** - adresa început zonă
- **întoarce** - TRUE pentru succes, FALSE altfel

Mapare memorie Win32

`LPVOID VirtualAlloc( LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect )` – alocă memorie în spatiul procesului curent

`LPVOID VirtualAllocEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect )` – alocă memorie în spațiul altui proces

- **hProcess** - handle proces pentru care se aplică funcția
- **fAllocationType** - MEM_RESERVE, MEM_COMMIT, MEM_RESET
- **lpAddress** - adresa unde începe alocarea; multiplu de 4KB pentru alocare și 64KB pentru rezervare; NULL - nicio preferință
- **dwSize** - dimensiunea zonei
- **flProtect** - modul de acces pentru zona alocată: PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_READONLY, PAGE_READWRITE, PAGE_WRITECOPY, PAGE_NOACCESS, PAGE_GUARD, PAGE_NOCACHE
- **întoarce** - pointer la zona alocată

`BOOL VirtualFree( LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType )` – eliberează zona de memorie din spațiul procesului curent

`BOOL VirtualFreeEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType )` – eliberează o zonă de memorie pentru alt proces

- **hProcess** -handle proces pentru care se aplică funcția
- **lpAddress, dwSize** - identifică zona
- **dwFreeType** - tipul operației: MEM_DECOMMIT, MEM_RELEASE
- **întoarce** - TRUE - succes, FALSE - eroare

`BOOL VirtualProtect( LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect )` – schimbarea protectiei unei zone de memorie mapate în spațiul procesului curent

`BOOL VirtualProtectEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect )` – schimbarea protecției unei zone de memorie mapate în alt proces

- **Process** - handle proces pentru care se aplică funcția
- **lpAddress, dwSize** - identifică zona
- **flNewProtect** - noi drepturi
- **lpflOldProtect** - salvare drepturi vechi
- **întoarce** - TRUE - succes, FALSE - eroare

Observație: functionează doar pentru pagini din aceeasi regiune rezervată alocată cu apelul VirtualAlloc sau VirtualAllocEx folosind MEM_RESERVE.

Interogarea zonelor mapate Win32

`DWORD VirtualQuery( LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer, SIZE_T dwLength )` – interogarea zonelor mapate din procesul curent `DWORD VirtualQueryEx( HANDLE hProcess, LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer, SIZE_T dwLength )` – interogarea zonelor mapate din alt proces

- **hProcess** - handle proces pentru care se aplică funcția
- **lpAddress** - adresa din cadrul zonei de interogat
- **lpBuffer** - buffer ce reține informații despre zonă
- **dwLength** - număr octeți scriși în buffer
- **întoarce** - 0 - nicio informație furnizată, diferit de 0 - altfel

Blocarea paginării Win32

`BOOL VirtualLock( LPVOID lpAddress, SIZE_T dwSize )` – blocare paginare proces curent

`BOOL VirtualLockEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize )` – blocare paginare alt proces

- **hProcess**handle proces pentru care se aplică funcția
- **lpAddress, dwSize**sunt luate in calcul paginile cu macar un octet în [lpAddress, lpAddess + dwSize]
- **întoarce**TRUE - succes, FALSE - altfel

`BOOL VirtualUnlock( LPVOID lpAddress, SIZE_T dwSize )` – repornirea paginării pentru procesul curent

`BOOL VirtualUnlockEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize )` – repornirea paginării pentru alt proces

Excepții Win32

`PVOID AddVectoredExceptionHandler( ULONG FirstHandler, PVECTORED_EXCEPTION_HANDLER VectoredHandler )` – adaugă o funcție de tratare excepții

- **firstHandler** - adaugare la început sau la final lista de tratat excepții
- **Vectored Handler** - funcția de tratat excepții

`ULONG RemoveVectoredExceptionHandler( PVOID VectoredHandlerHandle )` – elimină o funcție de tratat excepții

`LONG WINAPI VectoredHandler( PEXCEPTION_POINTERS ExceptionInfo )` – semnatura funcției de tratare a excepțiilor

struct _EXCEPTION_POINTERS: PEXCEPTION_RECORD ExceptionRecord; PCONTEXT ContextRecord;

struct _EXCEPTION_RECORD: DWORD ExceptionCode; DWORD ExceptionFlags; struct _EXCEPTION_RECORD* ExceptionRecord; PVOID ExceptionAddress; DWORD NumberParameters; ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];

- **ExceptionCode** va fi setat la EXCEPTION_ACCESS_VIOLATION sau EXCEPTION_DATATYPE_MISALIGNMENT
- **Exception Address** va fi setat la adresa instrucțiunii care a cauzat excepția
- **Number Parameters** va fi setat la 2
- **Exception Information** prima intrare e 0 pt citire, 1 pentru scriere; a doua intrare e adresa ce a generat excepția

SO Cheat Sheet
Memoria virtuală

Mapare fișiere și memorie POSIX

În POSIX trebuie incluse header-ele **sys/types.h**, **sys/mman.h**.

`void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)` – mapare fișier/memorie în spațiul de adresă al unui proces

`start`	adresa de start pentru mapare, NULL înseamna lipsa unei preferințe; dacă e diferită de NULL, e considerată doar un hint, maparea fiind creată la o adresă apropiată multiplu de dimensiunea unei pagini
`length`	lungimea mapării
`prot`	tipul de acces la zona de memorie: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
`flags`	tipul de mapare: cel puțin una din MAP_PRIVATE/MAP_SHARED, MAP_FIXED, MAP_LOCKED; MAP_ANONYMOUS pentru mapare memorie
`fd`	descriptorul fișierului mapat; ignorat la mapare memorie
`offset`	offset în cadrul fișierului mapat; ignorat la mapare memorie
întoarce	adresa mapării, MAP_FAILED (care reprezintă (void *) -1) în caz de eroare

`int msync(void *start, size_t length, int flags)` – declanșează în mod explicit sincronizarea fișierului cu maparea din memorie

`start, length`	identifică zona de memorie
`flags`	MS_SYNC, MS_ASYNC, MS_INVALIDATE
întoarce	0 în caz de succes, -1 în caz de eroare

`int munmap(void *start, size_t length)` – demapează o zonă din spațiul de adresă al procesului ; poate identifica zone aparținând unor mapari diferite, unele deja demapate

`void *mremap(void *old_address, size_t old_size, size_t new_size, unsigned long flags)` – redimensionare zonă mapată

`old_address, old_size`	identifică vechea zonă mapată
`new_size`	noua dimensiune
`flags`	MREMAP_MAYMOVEE
întoarce	pointer spre noua zonă în caz de succes, MAP_FAILED în caz de eroare

`int mprotect(const void *addr, size_t len, int prot)` – schimbă drepturile de acces ale unei mapări

`addr`	adresa mapării, multiplu de dimensiunea unei pagini
`len`	lungimea zonei considerate; se va aplica pentru toate paginile cu cel puțin un byte în intervalul [addr, addr + len -1]
`prot`	PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE
întoarce	0 în caz de succes, -1 în caz de eroare

`int madvise(void *start, size_t length, int advice)` – indicații despre cum va fi folosită o zonă mapată

`addr, length`	identifică zona
`advice`	MADV_NORMAL, MADV_RANDOM, MADV_SEQUENTIAL, MADV_WILLNEED, MADV_DONTNEED
întoarce	0 în caz de succes, -1 în caz de eroare

Blocarea paginării POSIX

`int mlock(const void *addr, size_t len)` – blochează paginarea paginilor incluse în intervalul [addr, addr + len - 1]

`int mlockall(int flags)` – blochează paginarea tuturor paginilor procesului

`flags`	MCL_CURRENT, MCL_FUTURE

`int munlock(const void *addr, size_t len)` – reporni paginarea tuturor paginilor din intervalul [addr, addr + len - 1]

`int munlockall(void)` – reporni paginarea tuturor paginilor procesului

Excepții accesare memorie POSIX

Funcția de tip sigaction va primi ca parametru o structură siginfo_t, având setate:

`si_signo`	SIGSEGV, SIGBUS
`si_code`	caz SIGSEGV: SEGV_MAPPER, SEGV_ACCERR; caz SIGBUS: BUS_ADRALN, BUS_ADRERR, BUS_OBJERR
`si_addr`	adresa care a generat excepția

ElectricFence

Folosit pentru depanare buffer overrun. Pentru a preveni și buffer underrun, definiți variabila de mediu EF_PROTECT_BELOW.

Programul trebuie linkat cu efence: -lefence.

Maparea fișierelor Win32

`HANDLE CreateFileMapping( HANDLE hFile, LPSECURITY_ATTRIBUTES lpAttributes, DWORD flProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR lpName )` – crează un obiect FileMapping, pentru a fi mapat

- **hFile** - handle fișier de mapat
- **lpAttributes** - atribute securitate pentru acces handle întors
- **flProtect** - tipul mapării: PAGE_READONLY, PAGE_READWRITE, PAGE_WRITECOPY
- **dwMaximumSizeHigh, dwMaximumSizeLow** - dimensiunea maximă
- **lpName** - șir identificare, opțional
- **întoarce** - în caz de succes întoarce un handle către un obiect FileMapping, în caz de eșec întoarce NULL

`LPVOID MapViewOfFile( HANDLE hFileMappingObject, DWORD dwDesiredAccess, DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, SIZE_T dwNumberOfBytesToMap )` – creează mapare fișier

- **hFileMappingObject** - obiect de tip FileMapping
- **dwDesiredAccess** - FILE_MAP_READ, FILE_MAP_WRITE, FILE_MAP_COPY
- **dwFileOffsetHigh, dwFileOffsetLow** - offset
- **dwNumberOfBytesToMap** - număr octeți de mapat
- **întoarce** - în caz de succes întoarce un pointer la zona mapată, în caz de eșec întoarce NULL

`BOOL UnmapViewOfFile( LPCVOID lpBaseAddress )` – demapare fișier mapat în memorie

- **lpBaseAddresst** - adresa început zonă
- **întoarce** - TRUE pentru succes, FALSE altfel

Mapare memorie Win32

`LPVOID VirtualAlloc( LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect )` – alocă memorie în spatiul procesului curent

`LPVOID VirtualAllocEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect )` – alocă memorie în spațiul altui proces

- **hProcess** - handle proces pentru care se aplică funcția
- **fAllocationType** - MEM_RESERVE, MEM_COMMIT, MEM_RESET
- **lpAddress** - adresa unde începe alocarea; multiplu de 4KB pentru alocare și 64KB pentru rezervare; NULL - nicio preferință
- **dwSize** - dimensiunea zonei
- **flProtect** - modul de acces pentru zona alocată: PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_READONLY, PAGE_READWRITE, PAGE_WRITECOPY, PAGE_NOACCESS, PAGE_GUARD, PAGE_NOCACHE
- **întoarce** - pointer la zona alocată

`BOOL VirtualFree( LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType )` – eliberează zona de memorie din spațiul procesului curent

`BOOL VirtualFreeEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType )` – eliberează o zonă de memorie pentru alt proces

- **hProcess** -handle proces pentru care se aplică funcția
- **lpAddress, dwSize** - identifică zona
- **dwFreeType** - tipul operației: MEM_DECOMMIT, MEM_RELEASE
- **întoarce** - TRUE - succes, FALSE - eroare

`BOOL VirtualProtect( LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect )` – schimbarea protectiei unei zone de memorie mapate în spațiul procesului curent

`BOOL VirtualProtectEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect )` – schimbarea protecției unei zone de memorie mapate în alt proces

- **Process** - handle proces pentru care se aplică funcția
- **lpAddress, dwSize** - identifică zona
- **flNewProtect** - noi drepturi
- **lpflOldProtect** - salvare drepturi vechi
- **întoarce** - TRUE - succes, FALSE - eroare

Observație: functionează doar pentru pagini din aceeasi regiune rezervată alocată cu apelul VirtualAlloc sau VirtualAllocEx folosind MEM_RESERVE.

Interogarea zonelor mapate Win32

`DWORD VirtualQuery( LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer, SIZE_T dwLength )` – interogarea zonelor mapate din procesul curent `DWORD VirtualQueryEx( HANDLE hProcess, LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer, SIZE_T dwLength )` – interogarea zonelor mapate din alt proces

- **hProcess** - handle proces pentru care se aplică funcția
- **lpAddress** - adresa din cadrul zonei de interogat
- **lpBuffer** - buffer ce reține informații despre zonă
- **dwLength** - număr octeți scriși în buffer
- **întoarce** - 0 - nicio informație furnizată, diferit de 0 - altfel

Blocarea paginării Win32

`BOOL VirtualLock( LPVOID lpAddress, SIZE_T dwSize )` – blocare paginare proces curent

`BOOL VirtualLockEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize )` – blocare paginare alt proces

- **hProcess**handle proces pentru care se aplică funcția
- **lpAddress, dwSize**sunt luate in calcul paginile cu macar un octet în [lpAddress, lpAddess + dwSize]
- **întoarce**TRUE - succes, FALSE - altfel

`BOOL VirtualUnlock( LPVOID lpAddress, SIZE_T dwSize )` – repornirea paginării pentru procesul curent

`BOOL VirtualUnlockEx( HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize )` – repornirea paginării pentru alt proces

Excepții Win32

`PVOID AddVectoredExceptionHandler( ULONG FirstHandler, PVECTORED_EXCEPTION_HANDLER VectoredHandler )` – adaugă o funcție de tratare excepții

- **firstHandler** - adaugare la început sau la final lista de tratat excepții
- **Vectored Handler** - funcția de tratat excepții

`ULONG RemoveVectoredExceptionHandler( PVOID VectoredHandlerHandle )` – elimină o funcție de tratat excepții

`LONG WINAPI VectoredHandler( PEXCEPTION_POINTERS ExceptionInfo )` – semnatura funcției de tratare a excepțiilor

struct _EXCEPTION_POINTERS: PEXCEPTION_RECORD ExceptionRecord; PCONTEXT ContextRecord;

struct _EXCEPTION_RECORD: DWORD ExceptionCode; DWORD ExceptionFlags; struct _EXCEPTION_RECORD* ExceptionRecord; PVOID ExceptionAddress; DWORD NumberParameters; ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];

- **ExceptionCode** va fi setat la EXCEPTION_ACCESS_VIOLATION sau EXCEPTION_DATATYPE_MISALIGNMENT
- **Exception Address** va fi setat la adresa instrucțiunii care a cauzat excepția
- **Number Parameters** va fi setat la 2
- **Exception Information** prima intrare e 0 pt citire, 1 pentru scriere; a doua intrare e adresa ce a generat excepția

SO Cheat Sheet
Thread-uri - Linux

Se va utiliza header-ul `pthread.h`

`int pthread_create(pthread_t *tid, const pthread_attr_t *tattr, void*(*routine)(void *), void *arg)` – creează un nou fir de execuţie

`tid`	identificatorul thread-ului
`tattr`	atributele noului thread (NULL - atribute implicite)
`routine`	specifică codul ce va fi executat de thread
`arg`	argumentele ce vor fi pasate funcţiei routine
întoarce	succes: 0, eroare: EAGAIN, EINVAL, EPERM

În caz de eroare, întoarce EAGAIN(nu există resursele necesare / PTHREAD_THREADS_MAX), EINVAL(tattr invalid), EPERM (eroare de permisiuni)

`int pthread_join(pthread_t th, void **th_return)` – suspendă execuţia thread-ului curent până când th termină

`th`	identificatorul thread-ului aşteptat
`th_return`	valoarea de return a thread-ului aşteptat
întoarce	succes: 0, eroare: EINVAL, ESRCH sau EDEADLK

`void pthread_exit(void *retval)` – termină un fir de execuţie

`retval`	valoarea de retur a thread-ului

`int pthread_key_create(pthread_key_t *key, void (*destr_func) (void *))` – creează o variabilă vizibilă tuturor threadurilor (fiecare thread va deţine valoarea specifică)

`key`	cheia variabilei
`destr_func`	dacă e diferită de NULL se va apela la terminarea thread-ului
întoarce	succes: 0, eroare: EAGAIN, ENOMEM

`int pthread_key_delete(pthread_key_t key)` – şterge o variabilă

`key`	cheia variabilei
întoarce	succes: 0, eroare: EINVAL

`int pthread_setspecific(pthread_key_t key, const void *pointer)` – modifică propria copie a variabilei

`key`	cheia variabilei
`pointer`	valoarea specifică ce trebuie stocată
întoarce	succes: 0, eroare: ENOMEM, EINVAL

`void* pthread_getspecific(pthread_key_t key)` – determină valoarea unei variabile de tip TSD

`key`	cheia
întoarce	valoarea specifică (NULL dacă nu e definită)

`void pthread_cleanup_push(void (*routine) (void *), void *arg)` – înregistrează o funcţie de cleanup

`routine`	rutina care va fi apelată
`arg`	argumentele corespunzătoare

`void pthread_cleanup_pop(int execute)` – deînregistrează o funcţie de cleanup

`execute`	doar dacă e diferit de 0 va executa și rutina

Trebuie inclus headerul **sched.h**

`int sched_yield(void)` – cedează dreptul de execuţie unui alt thread

Nu uitaţi să iniţializaţi :
pthread_once_t once_control = PTHREAD_ONCE_INIT;

`int pthread_once(pthread_once_t *once_control, void (*init_routine) (void))` – asigură că o bucată de cod (de obicei folosită pentru iniţializări) se execută o singură dată

`once_control`	pointer la o variabilă iniţializată cu PTHREAD_ONCE_INIT ce determină dacă init_routine a mai fost invocată
`init_routine`	invocată prima dată fără parametri
întoarce	succes: 0, eroare: EINVAL

`pthread_t pthread_self(void)` – determină identificatorul thread-ului curent

`int pthread_equal(pthread_t thread1, pthread_t thread2)` – determină dacă doi identificatori se referă la același thread

`thread1`	identificatorul pentru primul thread
`thread2`	identificatorul pentru al doilea thread
întoarce	o valoare diferită de 0 dacă sunt egali

`int pthread_setschedparam(pthread_t target_th, int policy, const struct sched_param *param)` – modifică priorităţile

`target_th`	poate fi SCHED_OTHER, SCHED_FIFO sau SCHED_RR
`param`	prioritatea pentru SCHED_FIFO sau SCHED_RR, în funcţie de implementare pentru SCHED_OTHER
întoarce	succes: 0, eroare: EINVAL, ENOTSUP, ENOTSUP, EPERM, EPERM , ESRCH

`int pthread_getschedparam(pthread_t target_th, int *policy, struct sched_param *param)` – află priorităţile

`target_th`	identificatorul thread-ului
`policy`	poate fi SCHED_OTHER, SCHED_FIFO sau SCHED_RR
`param`	prioritatea pentru SCHED_FIFO sau SCHED_RR, în funcţie de implementare pentru SCHED_OTHER
întoarce	succes: 0, eroare: ESRCH

Sincronizare

Mutex-uri

`PTHREAD_MUTEX_INITIALIZER` – macrodefiniţie pentru iniţializare

`int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)` – iniţializează un mutex cu atributele precizate

`mutex`	mutex-ul ce se doreşte iniţializat
`attr`	`NULL` sau iniţializat prin funcţiile `*mutexattr*`
întoarce	succes: 0

`int pthread_mutex_destroy(pthread_mutex_t *mutex)` – eliberează resursele alocate unui mutex

`int pthread_mutexattr_init(pthread_mutexattr_t *attr)` – iniţializează atributele unui mutex

`int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)` – eliberează atributele unui mutex

`int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)` – stabileşte comportamentul la preluarea recursivă a unui mutex

`attr`	atributele ce se doresc iniţializate
`type`	una din valorile • `PTHREAD_MUTEX_NORMAL` • `PTHREAD_MUTEX_ERRORCHECK` • `PTHREAD_MUTEX_RECURSIVE` • `PTHREAD_MUTEX_DEFAULT`
întoarce	succes: 0

`int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type)` – obţine comportamentul la preluarea recursivă a unui mutex

`int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)` – stabileşte modalitatea de moştenire a priorităţii de către un thread, la preluarea unui mutex

`attr`	atributele ce se doresc iniţializate
`protocol`	una din valorile • `PTHREAD_PRIO_NONE` • `PTHREAD_PRIO_INHERIT` • `PTHREAD_PRIO_PROTECT`
întoarce	succes: 0

`int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol)` – obţine modalitatea de moştenire a priorităţii de către un thread, la preluarea unui mutex

`int pthread_mutex_lock (pthread_mutex_t *mutex)` – ocupă, blocant, mutex-ul

`int pthread_mutex_trylock(pthread_mutex_t *mutex)` – încearcă ocuparea neblocantă a mutex-ului

`int pthread_mutex_unlock(pthread_mutex_t *mutex)` – eliberează mutex-ul

Semafoare

`sem_t* sem_open(const char *name, int oflag [, mode_t mode, unsigned int value])` – deschide un semafor cu nume, utilizat pentru sincronizarea mai multor procese

`name`	identifică semaforul
`oflag`	`O_CREAT` sau/și `O_EXCL`
`mode`	specifică permisiunile noului semafor
`value`	valoarea inițială
întoarce	adresa semaforului

`int sem_close(sem_t *sem)` – închide un semafor cu nume

`int sem_unlink(const char *name)` – înlătură din sistem un semafor cu nume

`int sem_init(sem_t *sem, int pshared, unsigned int value)` – deschide un semafor fără nume

`sem`	adresa semaforului
`pshared`	0, dacă este folosit în cadrul unui singur proces, SAU nenul, dacă este folosit pentru sincronizarea unor procese diferite, caz în care trebuie alocat într-o zonă de memorie partajată
`value`	valoarea inițială
întoarce	succes: 0

`int sem_destroy(sem_t *sem)` – distruge un semafor fără nume

`int sem_post(sem_t *sem)` – incrementează semaforul

`int sem_wait(sem_t *sem)` – decrementează, blocant, semaforul

`int sem_trywait(sem_t *sem)` – decrementează, neblocant, semaforul

`int sem_getvalue(sem_t *sem, int *pvalue)` – obţine valoarea semaforului

Variabile de condiţie

`PTHREAD_COND_INITIALIZER` – macrodefiniţie pentru iniţializare

`int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr)` – iniţializează o variabilă de condiţie cu atributele precizate

`cond`	variabila de condiţie ce se doreşte iniţializată
`attr`	`NULL` sau iniţializat prin funcţiile `*condattr*`
întoarce	succes: 0

`int pthread_cond_destroy(pthread_cond_t *cond)` – eliberează resursele alocate unei variabile de condiţie

`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)` – suspendă execuţia firului apelant, eliberând, atomic, mutex-ul asociat

`cond`	variabila de condiţie la care se suspendă firul apelant
`mutex`	mutex-ul asociat
întoarce	succes: 0

`int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)` – suspendă execuţia firului apelant, nu mai mult de un interval specificat, eliberând, atomic, mutex-ul asociat

`cond`	variabila de condiţie la care se suspendă firul apelant
`mutex`	mutex-ul asociat
`abstime`	perioada maxima de suspendare
întoarce	succes: 0, eroare: -1 cu eroarea `ETIMEDOUT`, în cazul în care expiră timeout-ul

`int pthread_cond_signal(pthread_cond_t *cond)` – semnalizează un fir suspendat la variabila de condiţie

`int pthread_cond_broadcast(pthread_cond_t *cond)` – semnalizează toate firele suspendate la variabila de condiţie

Bariere

Pentru lucrul cu bariere este necesară definirea macro-ului `_XOPEN_SOURCE` la o valoare de cel puţin 600.

`int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned count)` – iniţializează o barieră cu atributele precizate

`barrier`	bariera ce se doreşte iniţializată
`attr`	`NULL` sau iniţializat prin funcţiile `*barrierattr*`
`count`	numărul de fire de execuţie care trebuie să ajungă la barieră pentru ca aceasta să fie eliberată
întoarce	succes: 0

`int pthread_barrier_destroy(pthread_barrier_t *barrier)` – eliberează resursele alocate barierei

`int pthread_barrier_wait(pthread_barrier_t *barrier)` – suspendă primele `count-1` fire care o apelează, acestea fiind trezite la apelul cu numărul `count`

`barrier`	bariera la care se realizează aşteptarea
întoarce	success: PTHREAD_BARRIER_SERIAL_THREAD, 0, eroare: EINVAL

SO Cheat Sheet
Thread-uri - Windows

`HANDLE CreateThread (LPSECURITY_ATTRIBUTES lpThAttr, SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpParameter, DWORD dwCreatFlags, LPDWORD lpThreadId)` – creează un fir de execuţie

- **lpThAttr** - pointer la o structură de tip SECURITY_ATTRIBUTES, dacă e NULL handle-ul nu poate fi moștenit
- **dwStackSize** - mărimea iniţială a stivei, în bytes; 0 - mărimea implicită
- **lpStartAddr** - pointer la funcţia ce trebuie executată
- **lpParameter** - opţional - pointer la o variabilă
- **dwCreatFlags** - opţiuni: 0, CREATE_SUSPENDED, STACK_SIZE_PARAM_IS_A_RESERVATION
- **lpThreadId** - pointer unde va fi întors identificatorul
- **întoarce** - handle către threadul creat

`HANDLE OpenThread(DWORD dwDesireAccess, BOOL bInheritHandle, DWORD dwThreadId)` – deschide un obiect de tip Thread existent

- **dwDesireAccess** - drepturile de acces
- **bInheritHandle** - TRUE - handle-ul poate fi mostenit
- **dwThreadId** - identificatorul thread-ului
- **întoarce** - handle către thread

`DWORD WaitForSingleObject(HANDLE hHANDLE, DWORD dwMilliseconds)` – aşteptă terminarea unui fir de execuţie

- **hHandle** - handle către obiect
- **dwMilliseconds** - intervalul de timeout(0 până la INFINITE)

`void ExitThread(DWORD dwExitCode)` – terminarea threadului curent cu specificarea codului de terminare

`BOOL TerminateThread(HANDLE hThread, DWORD dwExitCode)` – terminarea unui thread hThread cu specificarea codului de terminare

`BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode)`

- **hThread** - handle-ul threadului ce trebuie să aibă dreptul de acces THREAD_QUERY_INFORMATION
- **lpExitCode** - pointer la o variabilă în care va fi plasat codul de terminare al firului. Dacă firul nu și-a terminat execuţia, această valoare va fi STILL_ACTIVE

`DWORD SuspendThread(HANDLE hThread)` – suspendă execuţia unui thread

`DWORD ResumeThread(HANDLE hThread)` – reia execuţia unui thread

Aceste funcţii nu pot fi folosite pentru sincronizare, dar sunt utile pentru programe de debug.

`void Sleep(DWORD dwMilliseconds)` – suspendă execuţia unui thread pe o perioadă de dwMilliseconds

`BOOL SwitchToThread(void)` – firul de execuţie renunţă doar la timpul pe care îl avea pe procesor în momentul respectiv

întoarce TRUE dacă procesorul este cedat unui alt thread şi FALSE dacă nu există alte thread-uri gata de execuţie

`HANDLE GetCurrentThread(void)` – întoarce un pseudo-handle pentru firul curent ce nu poate fi folosit decât de firul apelant

`DWORD GetCurrentThreadId(void)` – întoarce identificatorul threadului

`DWORD GetThreadId(HANDLE Thread)` – întoarce identificatorul threadului ce corespunde handle-ului Thread

Sincronizare

Funcţii de asteptare

Funcţiile de aşteptare sunt cele din familia `WaitForSingleObject` şi au fost prezentate, în detaliu, în laboratorul de comunicaţie interproces.

Mutex Win32

Funcţiile de mai jos au fost prezentate, în detaliu, în cadrul laboratorului de comunicaţie interproces.

`HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttr, BOOL bInitialOwner, LPCTSTR lpName)` – creează un mutex

`HANDLE OpenMutex(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)` – deschide un mutex

`BOOL ReleaseMutex(HANDLE hMutex)` – cedează posesia mutexului

Semafoare Win32

Funcţiile de mai jos au fost prezentate, în detaliu, în cadrul laboratorului de comunicaţie interproces.

`HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpSemAttr, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpNAME)` – creează/deschide un semafor

`HANDLE OpenSemaphore(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpNAME)` – deschide un semafor existent

`BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount, LPLONG lpPreviousCount)` – incrementează semaforul

Evenimente

`HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpEventAttributes, BOOL bManualReset, BOOL bInitialState, LPCTSTR lpName)` – creează un eveniment

- **lpEventAttributes** - pointer la o structură de tip SECURITY_ATTRIBUTES, dacă e NULL handle-ul nu poate fi moștenit
- **bManualReset** - TRUE - pentru manual-reset, FALSE - auto-reset
- **bInitialState** - TRUE - evenimentul e creat în starea signaled
- **lpName** - numele evenimentului sau NULL dacă se dorește a fi anonim
- **întoarce** - handle către evenimentul creat

`HANDLE SetEvent(HANDLE hEvent)` – setează evenimentul în starea signaled

`HANDLE ResetEvent(HANDLE hEvent)` – setează evenimentul în starea non-signaled

`HANDLE PulseEvent(HANDLE hEvent)` – semnalează toate thread-urile care așteaptă la un eveniment manual-reset

Secţiuni critice

`void InitializeCriticalSection(LPCRITICAL_SECTION pcrit_sect)` – iniţializează o secțiune critică cu un contor de spin implicit egal cu 0

`BOOL InitializeCriticalSectionAndSpinCount(LPCRITICAL_SECTION pcrit_sect, DWORD dwSpinCount)` – permite specificarea contorului de spin

`DWORD SetCriticalSectionSpinCount(LPCRITICAL_SECTION pcrit_sect, DWORD dwSpinCount)` – permite modificarea contorului de spin al unei secțiuni critice

`void DeleteCriticalSection(LPCRITICAL_SECTION pcrit_sect)` – distruge o secțiune critică

`void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)` – similar cu `pthread_mutex_lock()` pentru mutexuri `RECURSIVE`

`void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)` – similar cu `pthread_mutex_unlock()` pentru mutexuri `RECURSIVE`

`BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCritSect)` – similar cu `pthread_mutex_trylock()` pentru mutexuri `RECURSIVE`

`lpCritSect` secţiunea critică

întoarce `FALSE` dacă secțiunea critică este ocupată de alt fir de execuție

Interlocked Variable Access

`LONG InterlockedIncrement(LONG volatile *lpAddend)` – incrementează, atomic, valoarea indicată de `lpAddend` și întoarce valoarea incrementată

`LONG InterlockedDecrement(LONG volatile *lpDecend)` – decrementează, atomic, valoarea indicată de `lpAddend` și întoarce valoarea decrementată

`LONG InterlockedExchange(LONG volatile *Target, LONG Value)` – scrie valoarea întreagă `Value` în zona indicată de `Target` şi întoarce vechea valoarea a lui `Target`

`LONG InterlockedExchangeAdd(LPLONG volatile Addend, LONG Value)` – adaugă valoarea întreagă `Value` la zona indicată de `Addend` şi întoarce vechea valoarea a lui `Addend`

`PVOID InterlockedExchangePointer(PVOID volatile *Target, PVOID Value)` – atribuie pointerul `Value` pointerului indicat de pointerul `Target`

`LONG InterlockedCompareExchange(LONG volatile *dest, LONG exchange, LONG comp)` – compară valoarea de la adresa `dest` cu valoarea `comp` şi, dacă sunt egale, setează atomic valoarea de la adresa `dest` la valoarea `exchange`

`PVOID InterlockedCompareExchangePointer(PVOID volatile *dest, PVOID exchange, PVOID comp)` – compară pointerul de la adresa `dest` cu pointerul `comp` şi, dacă sunt egale, setează atomic pointerul de la adresa `dest` la pointerul `exchange`

Timer Queues

`HANDLE CreateTimerQueue(void)` – întoarce un handle la coada de timere

`BOOL DeleteTimerQueue(HANDLE TimerQueue)` – marchează coada pentru ștergere, dar NU așteaptă ca toate callbackurile asociate cozii să se termine

`BOOL DeleteTimerQueueEx(HANDLE TimerQueue, HANDLE CompletionEv)` – eliberează resursele alocate cozii, oferind facilităţi suplimentare

- **TimerQueue** - coada
- **CompletionEv** - una din valorile `NULL`, `INVALID_HANDLE_VALUE` SAU un handle de tip `Event`
- **întoarce** - non-zero pentru succes

`BOOL CreateTimerQueueTimer(PHANDLE phNewTimer, HANDLE TimerQueue, WAITORTIMERCALLBACK Callback, PVOID Parameter, DWORD DueTime, DWORD Period, ULONG Flags)` – creează un timer

- **phNewTimer** - aici întoarce un `HANDLE` la timerul nou creat
- **TimerQueue** - coada la care este adăugat timerul. Dacă e `NULL` se folosește o coadă implicită
- **Callback** - callback de executat
- **Parameter** - parametru trimis callbackului
- **DueTime** - intervalul, în milisecunde, după care va expira, prima dată, timerul
- **Period** - perioada, în milisecunde, a timerului
- **Flags** - tipul callbackului: IO/NonIO, `EXECUTEONLYONCE`
- **întoarce** - non-zero pentru succes

`VOID WaitOrTimerCallback(PVOID lpParameter, BOOLEAN TimerOrWaitFired)` – semnătura unui callback

`BOOL ChangeTimerQueueTimer(HANDLE TimerQueue, HANDLE Timer, ULONG DueTime, ULONG Period)` – modifică timpul de expirare a unui timer

`BOOL CancelTimerQueueTimer(HANDLE TimerQueue, HANDLE Timer)` – dezactivează unui timer

`BOOL DeleteTimerQueueTimer(HANDLE TimerQueue, HANDLE Timer, HANDLE CompletionEvent)` – dezactivează ȘI distruge unui timer. `CompletionEvent` e similar cu cel din `DeleteTimerQueueEx`.

Registered Wait Functions

`BOOL RegisterWaitForSingleObject(PHANDLE phNewWaitObject, HANDLE hObject, WAITORTIMERCALLBACK Callback, PVOID Context, ULONG dwMilliseconds, ULONG dwFlags)` – înregistrează în thread pool o funcție de așteptare, al cărei callback va fi executat când expiră timeout-ul SAU când obiectul la care se aşteaptă, `hObject`, trece în starea `SIGNALED`

- **phNewWaitObject** - aici întoarce un `HANDLE` la timerul nou creat
- **hObject** - obiectul de sincronizare la care se aşteaptă
- **Callback** - callback de executat
- **Context** - parametru trimis callbackului
- **dwMilliseconds** - timeout
- **dwFlags** - `EXECUTEONLYONCE` etc.
- **întoarce** - non-zero pentru succes

`VOID CALLBACK WaitOrTimerCallback(PVOID lpParameter, BOOLEAN TimerOrWaitFired)` – semnătura unui callback

`BOOL UnregisterWait(HANDLE WaitHandle)` – anulează înregistrarea unei funcții de așteptare

`BOOL UnregisterWaitEx(HANDLE WaitHandle, HANDLE CompletionEv)` – anulează înregistrarea unei funcții de așteptare

- **WaitHandle** - handle-ul funcţiei
- **CompletionEv** - asemănător cu parametrul lui `DeleteTimerQueueEx`
- **întoarce** - non-zero pentru succes

Operaţii I/O avansate

Windows

Operaţii overlapped

`struct OVERLAPPED`

- **Internal**, codul de eroare pentru cererea de I/O
- **InternalHigh**, numărul de octeţi transferaţi
- **(Offset,OffsetHigh)**, offset de început al operaţiei de I/O
- **hEvent**, eveniment ce va fi semnalat la terminarea operaţiei de I/O

`FILE_FLAG_OVERLAPPED`

- **atribut** asociat unui HANDLE pentru operaţii overlapped
- **CreateFile("myfile.txt",**
  GENERIC_READ,
  0,
  NULL,
  OPEN_EXISTING,
  FILE_FLAG_OVERLAPPED,
  NULL)

`GetOverlappedResult(hFile, lpOverlapped, bytesTransferred, bWait`

- **hFile**, file handle
- **lpOverlapped**, structura overlapped specificată la lansarea operaţiei I/O
- **bytesTransferred**, numărul de octeţi transferaţi
- **bWait**, dacă este TRUE apelul întoarce doar atunci când operaţia de I/O s-a încheiat.
  Dacă este FALSE şi operaţia încă nu s-a încheiat funcţia întoarce ERROR_IO_INCOMPLETE

Completion Ports

`HANDLE CreateIoCompletionPort(fileHandle, existingCompletionPort, completionKey, noThreads)`

- **creare completion port**
  - **fileHandle**, INVALID_HANDLE_VALUE
  - **existingCompletionPort**, NULL
  - **completionKey**, NULL
  - **noThreads**, numărul maxim de threaduri care pot rula concurent. Dacă acest parametru este 0 numărul maxim va fi egal cu numărul procesoarelor din sistem
- **adăugare file handle la completion port**
  - **fileHandle**, overlapped handle care va fi adăugat la completion port
  - **existingCompletionPort**, completion port creat anterior
  - **completionKey**, pointer către o zonă de memorie care va identifica operaţia de I/O
  - **noThreads**, în acest caz valoarea este ignorată

`BOOL GetQueuedCompletionStatus(CompletionPort, bytesTransferred, lpCompletionKey, lpOverlapped, dwMiliseconds)`

- **CompletionPort**, handle către completion port
- **bytesTransferred**, numărul octeţilor transferaţi în timpul unei operaţii de I/O încheiate
- **lpCompletionKey**, întoarce pointer către cheia asociată cu file handle-ul pentru care I/O s-a încheiat
- **lpOverlapped**, întoarce pointer către structura overlapped dată ca parametru operaţiei de I/O
- **dwMiliseconds**, timeout pentru care se aşteaptă ca o operaţie să se încheie. Pentru apel fără timeout se foloseşte INFINITE.

Zero Copy

`TransmitFile(hSocket, hFile, numberOfBytesToWrite, numberOfBytesPerSend, lpOverlapped, lpTransmitBuffers, dwFlags`

- **hSocket**, handle către un socket conectat
- **hFile**, handle către fişierul de transmis
- **numberOfBytesToWrite**, numărul de octeţi din fişier de transmis
- **numberOfBytesPerSend**, dimensiunea în octeţi pentru fiecare bloc de date trimis
- **lpOverlapped**, pointer către o structură overlapped care va declanşa un apel asincron
- **lpTransmitBuffers**, NULL
- **dwFlags**, 0

SO Cheat Sheet

Operatii I/O avansate - Linux

Multiplexare IO

select

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout)
```

`nfds`	valoarea celui mai mare file descriptor plus 1
`readfds`	file descriptoriii urmăriți pentru citire
`writefds`	file descriptorii urmăriți pentru scriere
`exceptfd`	file descriptorii urmăriți pentru excepții
`timeout`	timpul maxim după care select se întoarce. NULL semnifică blocarea indefinit
întoarce	numărul total de file descriptori urmăriți; 0 dacă timeout-ul a expirat sau -1 în caz de eroare

poll

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout)
```

`fds`	conține un descriptor de fișier, evenimentele urmărite pe acest descriptor și parametrul de ieșire care ne spune dacă a apărut un eveniment pe acel descriptor
`nfds`	numărul strcturilor fds
`timeout`	timpul maxim după care poll se întoarce. -1 semnifică blocarea indefinit
întoarce	numărul de structuri pentru care au apărut evenimente; 0 dacă timeout-ul a expirat sau -1 în caz de eroare

epoll

```
int epoll_create(int size)
```

`size`	hint pentru kernel asupra numărului de descriptori ce vor fi urmăriți
întoarce	un file descriptor sau -1 în caz de eroare

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event
*event)
```

`epfd`	file descriptor obținut în urma unui apel epoll_create
`op`	operația efectuată asupra epfd. Poate fi una dintre: EPOLL_CTL_ADD, EPOLL_CTLR_DEL, EPOLL_CTL_MOD
`fd`	file descriptor pentru care se face operația op
`event`	structura care descrie evenimentul urmărit
întoarce	0 pentru succes; -1 în caz de eroare

```
int epoll_wait(int epfd, struct epoll_event *event, int
max_events, int timeout)
```

`epfd`	file descriptor obținut în urma unui apel epoll_create
`events`	parametru de ieșire în care se vor pune evenimentele disponibile; trebuie să fie prealocat
`max_events`	numărul maxim de eventimente întoarse
`timeout`	timpul maxim după care funcția se întoarce; -1 pentru așteptare la infinit
întoarce	numărul de file descriptori disponibili pentru I/O sau -1 în caz de eroare

Generalizarea Multiplexării

```
int eventfd(unsigned int initval, int flags)
```

`initval`	valoarea inițială a contorului intern
`flags`	flaguri pentru a schimba comportamentul lui eventfd; poate fi lăsat 0
întoarce	file descritpro eventfd sau -1 în caz de eroare

```
int signalfd(int fd, const sigset_t *mask, int flags)
```

`fd`	-1 pentru a crea un nou descriptor signalfd, sau un descriptor deja existent pentru modificarea măștii
`mask`	masca de semnale pe care apelantul dorește să le accepte via descriptorul de fișier
`flags`	flaguri pentru a schimba comportamentul lui signalfd; poate fi lăsat 0
întoarce	file descriptor sau -1 în caz de eroare

Operații asincrone

```
void io_prep_pread(struct iocb *iocb, int fd, void *buf,
size_t count, long long offset)
```

`iocb`	structura iocb care va fi inițializată
`fd`	file descriptorul pe care se va face operația
`count`	cât se dorește să fie scris
`offset`	offsetul din fișier de unde să aibă loc operația
întoarce	nimic

```
void io_prep_pwrite(struct iocb *iocb, int fd, void *buf,
size_t count, long long offset)
```

`iocb`	structura iocb care va fi inițializată
`fd`	file descriptorul pe care se va face operația
`count`	cât se dorește să fie citit
`offset`	offsetul din fișier de unde să aibă loc operația
întoarce	nimic

```
int io_setup(unsigned int nr_events, io_context_t *ctx)
```

`nr_events`	numărul de evenimente care pot fi primite în contextul curent
`ctx`	parametru de ieșire în care va fi salvat noul context io
întoarce	0 pentru succes sau -1 în caz de eroare

```
int io_destroy(io_context_t *ctx)
```

`ctx`	context AIO deja existent
întoarce	0 pentru succes sau -1 în caz de eroare

```
int io_submit(io_context_t ctx, long nr, struct iocb *ios[])
```

`ctx`	context create anterior
`nr`	numărul de elemente din vectorul ios
`ios`	vector de pointeri la structurile iocb în care se află operațiile care se doresc a fi submise
întoarce	numărul de structuri iocb submise(poate fi și 0); în caz de eroare întoarce un număr negativ care desemnează eroarea

```
int io_getevent(io_context_t ctx, long min_nr, long nr,
struct io_event *events, struct timespec *timeout)
```

`ctx`	context AIO deja existent
`min_nr`	numărul minim de evenimente care trebuie întoarse
`nr`	numărul maxim de evenimente care trebuie întoarse
`events`	vector de evenimente cu evenimentele întoarse
`timeout`	specifică cât să aștepte operația; NULL înseamnă că operația se va întoarce pentru min_nr evenimente dacă operația are success
întoarce	numărul de evenimente terminate până la timeout (poate fi și 0) sau un număr negativ reprezentând codul de eroare

```
int io_cancel(io_context_t ctx, struct iocb *iocb,struct
io_event *evt)
```

`ctx`	context create anterior
`iocb`	structura iocb corespunzând operațieicare se dorește a fi anulate
`result`	dacă există deja un rezultat, va fi întors în aceasta variabilă
întoarce	0 în caz de succes; în caz de eroare întoarce un număr negativ care desemnează eroarea

Vectored IO

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt)
```

`fd`	file descriptor pe care se face opearția
`iov`	vector cu structuri reprezentând bufferele din care se va citi
`iovcnt`	numărul de elemente ale vectorului iov
întoarce	numărul de octeți citiți sau -1 în caz de eroare

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt)
```

`fd`	file descriptor pe care se face opearția
`iov`	vector cu structuri reprezentând bufferele din care se va scrie
`iovcnt`	numărul de elemente ale vectorului iov
întoarce	numărul de octeți scriși sau -1 în caz de eroare

SO Cheat Sheet
Profiling

gprof

- **-pg**, la compilare
- **./executbil** rulează comanda
- **gprof ./executabil** citirea și interpretarea datelor

oprofile

- **ophelp** lista de evenimente posibile
- **opcontrol –init** inserarea (pornirea) modului oprofile
- **opcontrol –event=TIP_EVENIMENT:COUNT:UNIT-MASK:KERNEL-SPACE-COUNTING:USER-SPACE-COUNTING** configurarea evenimentelor
  - **TIP_EVENIMENT** - reprezintă numărul evenimentului care se dorește monitorizat
  - **COUNT** - reprezintă numărul de evenimente hardware de acest tip după care procesorul emite o întrerupere NMI.
  - **UNIT-MASK** - o valoare numerică ce poate alterează comportamentul pentru counterul curent. Dacă nu este specificat se folosește o valoarea implicită (poate fi determinată cu ophelp).
  - **KERNEL-SPACE-COUNTING** - poate lua două valori, 0/1, și specifică dacă se activează sau nu counterul atunci când se rulează cod kernel. Implicit are valoarea 1.
  - **USER-SPACE-COUNTING** - poate lua două valori, 0/1, și specifică dacă se activează sau nu counterul atunci când se rulează cod user. Implicit are valoarea 1.
- **opcontrol –status** aflarea configurării curente
- **opcontrol –no-vmlinux –image=./nume_binar** setarea unei imagini care va fi eșantionată
- **opcontrol –start** pornirea daemonului care va realiza eșantionarea
- **opcontrol –dump; opcontrol –shutdown** salvarea datelor și oprirea daemonului
- **opreport** interpretarea datelor salvate
- **opannotate –source** adnotarea surselor cu informații