C Sistem ve Terminal Fonksiyonları Detaylı Rehberi

Readline Kütüphanesi Fonksiyonları

`readline`

```c
char *readline(const char *prompt);
```

Parametreler:

- `prompt`: Kullanıcıya gösterilecek komut istemi metni (örn: "$ ", "minishell> ")

İşlevi:

- Kullanıcıdan interaktif bir satır okur
- Otomatik tamamlama, geçmiş gezinme (↑↓ tuşları) destekler
- Satır düzenleme özelliği sağlar (Ctrl+A başa git, Ctrl+E sona git)

Dönüş: Okunan satırın malloc ile ayrılmış kopyası, EOF durumunda NULL

`rl_clear_history`

```c
void rl_clear_history(void);
```

Parametreler: Yok

İşlevi:

- Readline'ın sakladığı tüm komut geçmişini siler
- Bellek temizliği yapar
- Yeni oturum başlarken kullanılır

`rl_on_new_line`

```c
int rl_on_new_line(void);
```

Parametreler: Yok

İşlevi:

- Readline'a cursor'un yeni bir satırda olduğunu bildirir
- Terminal durumu senkronizasyonu sağlar
- Prompt yeniden çizimi öncesi kullanılır

`rl_replace_line`

```c
void rl_replace_line(const char *text, int clear_undo);
```

Parametreler:

- `text`: Mevcut satırı değiştirecek yeni metin
- `clear_undo`: 1 ise undo geçmişini temizler, 0 ise korur

İşlevi:

- Mevcut düzenleme satırını tamamen yeni metinle değiştirir
- Otomatik düzeltme veya makro genişletme için kullanılır

`rl_redisplay`

```c
void rl_redisplay(void);
```

Parametreler: Yok

İşlevi:

- Mevcut satırı terminalde yeniden çizer
- Terminal bozulması veya signal sonrası düzeltme için
- Ekran güncellemesi yapar

`add_history`

```c
void add_history(const char *line);
```

Parametreler:

- `line`: Geçmişe eklenecek komut satırı

İşlevi:

- Verilen satırı readline geçmişine ekler
- ↑↓ tuşları ile erişilebilir hale getirir
- Genelde boş satırlar eklenmez

Temel C Kütüphane Fonksiyonları

`printf`

```c
int printf(const char *format, ...);
```

Parametreler:

- `format`: Format string'i (%d, %s, %c, %x, %f vb. içerir)
- `...`: Format'a göre değişken sayıda argüman

Format Belirteçleri:

- `%d` / `%i`: int (decimal)
- `%s`: string (char *)
- `%c`: char
- `%x`: hexadecimal (küçük harf)
- `%X`: hexadecimal (büyük harf)
- `%f`: float/double
- `%p`: pointer adresi

Dönüş: Yazdırılan karakter sayısı, hata durumunda negatif

`malloc`

```c
void *malloc(size_t size);
```

Parametreler:

- `size`: Ayrılacak bellek miktarı (byte cinsinden)

İşlevi:

- Heap'ten dinamik bellek ayırır
- Başlangıç değeri tanımsız (garbage value)

- Başarısızlık durumunda NULL döner

Dönüş: Ayrılan bellek alanının başlangıç adresi veya NULL

`free`

```c
void free(void *ptr);
```

Parametreler:

- `ptr`: Daha önce malloc/calloc/realloc ile ayrılan bellek adresi

İşlevi:

- Dinamik olarak ayrılan belleği sisteme geri verir
- ptr NULL ise hiçbir şey yapmaz
- Double free undefined behavior'a neden olur

Dosya ve I/O İşlemleri

`write`

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

Parametreler:

- `fd`: Dosya tanımlayıcısı (0=stdin, 1=stdout, 2=stderr)
- `buf`: Yazılacak verinin bulunduğu buffer
- `count`: Yazılacak byte sayısı

İşlevi:

- Sistem çağrısı, buffering yapmaz
- Doğrudan kernel'e gider

Dönüş: Yazılan byte sayısı, hata durumunda -1

`access`

```c
int access(const char *pathname, int mode);
```

Parametrelers:

- `pathname`: Kontrol edilecek dosya yolu
- `mode`: Kontrol modu (R_OK, W_OK, X_OK, F_OK)

Mod Sabitleri:

- `F_OK`: Dosya var mı?
- `R_OK`: Okuma izni var mı?
- `W_OK`: Yazma izni var mı?
- `X_OK`: Çalıştırma izni var mı?

Dönüş: 0 başarı, -1 hata

`open`

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

Parametreler:

- `pathname`: Açılacak dosya yolu
- `flags`: Açma modu bayrakları
- `mode`: Yeni dosya oluşturulursa izinleri (sadece O_CREAT ile)

Flag Sabitleri:

- `O_RDONLY`: Sadece okuma
- `O_WRONLY`: Sadece yazma
- `O_RDWR`: Okuma ve yazma
- `O_CREAT`: Yoksa oluştur
- `O_TRUNC`: Mevcut içeriği sil
- `O_APPEND`: Dosya sonuna ekle

Mode Sabitleri:

- `0644`: rw-r--r-- (owner okuma/yazma, diğerleri okuma)
- `0755`: rwxr-xr-x (owner tam, diğerleri okuma/çalıştırma)

Dönüş: Dosya tanımlayıcısı, hata durumunda -1

`read`

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

Parametreler:

- `fd`: Dosya tanımlayıcısı
- `buf`: Okunan verinin saklanacağı buffer
- `count`: Okunacak maksimum byte sayısı

İşlevi:

- Dosyadan veri okur
- EOF'ta 0 döner
- Kısmi okuma olabilir (requested < actual)

Dönüş: Okunan byte sayısı, 0 EOF, -1 hata

`close`

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

Parametreler:

- `fd`: Kapatılacak dosya tanımlayıcısı

İşlevi:

- Dosya tanımlayıcısını kapatır
- Sistem kaynaklarını serbest bırakır
- Aynı fd'yi tekrar kullanmaya çalışmak hata

Dönüş: 0 başarı, -1 hata

Process Yönetimi

`fork`

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

Parametreler: Yok

İşlevi:

- Mevcut process'in tam kopyasını oluşturur
- Parent ve child process'leri oluşur
- Copy-on-write optimizasyonu kullanılır

Dönüş:

- Parent'ta: child'ın PID'si
- Child'ta: 0
- Hata durumunda: -1

`wait`

```c
pid_t wait(int *wstatus);
```

Parametreler:

- `wstatus`: Child'ın çıkış durumu bilgisi (NULL olabilir)

İşlevi:

- Herhangi bir child process'in bitmesini bekler
- Child'ın zombie durumunu temizler
- Blocking çağrı

Dönüş: Biten child'ın PID'si, hata durumunda -1

`waitpid`

```c
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Parametreler:

- `pid`: Beklenecek specific child PID'si (-1 herhangi biri için)
- `wstatus`: Çıkış durumu bilgisi
- `options`: Bekleme seçenekleri (WNOHANG, WUNTRACED)

Seçenekler:

- `WNOHANG`: Non-blocking, child bitmemişse hemen dön
- `WUNTRACED`: Durdurulmuş child'ları da raporla

Dönüş: Child PID'si, 0 (WNOHANG ile), -1 hata

`wait3` / `wait4`

```c
pid_t wait3(int *wstatus, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *wstatus, int options, struct rusage *rusage);
```

Ek Parametreler:

- `rusage`: Kaynak kullanım istatistikleri (CPU time, memory vb.)

İşlevi:

- wait/waitpid'in genişletilmiş versiyonları
- Detaylı kaynak kullanım bilgisi sağlar

Signal İşlemleri

`signal`

```c
sighandler_t signal(int signum, sighandler_t handler);
```

Parametreler:

- `signum`: Signal numarası (SIGINT, SIGTERM, SIGKILL vb.)
- `handler`: Signal handler fonksiyonu veya SIG_DFL/SIG_IGN

Signal Türleri:

- `SIGINT`: Interrupt (Ctrl+C)
- `SIGTERM`: Terminate
- `SIGKILL`: Kill (yakalanamaz)
- `SIGCHLD`: Child process bitti
- `SIGPIPE`: Broken pipe

Dönüş: Önceki handler, SIG_ERR hata durumunda

`sigaction`

c

```c
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Parametreler:

- `signum`: Signal numarası
- `act`: Yeni signal action yapısı
- `oldact`: Önceki action'ı saklamak için (NULL olabilir)

struct sigaction alanları:

- `sa_handler`: Signal handler fonksiyonu
- `sa_mask`: Block edilecek signal'lar
- `sa_flags`: Davranış bayrakları (SA_RESTART vb.)

İşlevi:

- signal() fonksiyonunun gelişmiş versiyonu
- Daha fazla kontrol ve taşınabilirlik sağlar

`sigemptyset`

c

```c
int sigemptyset(sigset_t *set);
```

Parametreler:

- `set`: Boşaltılacak signal set'i

İşlevi:

- Signal set'ini boşaltır (hiçbir signal içermez)
- sigaction ile kullanılmadan önce initialize eder

`sigaddset`

c

```c
int sigaddset(sigset_t *set, int signum);
```

Parametreler:

- `set`: Signal set'i

- `signum`: Eklenecek signal

İşlevi:

- Belirtilen signal'ı set'e ekler
- sigaction'da sa_mask ile kullanılır

`kill`

```c
int kill(pid_t pid, int sig);
```

Parametreler:

- `pid`: Hedef process PID'si (0, -1 özel anlamlar)
- `sig`: Gönderilecek signal

PID Değerleri:

- `> 0`: Specific process'e gönder
- `0`: Aynı process group'taki tüm process'lere
- `-1`: Tüm process'lere (yetkili ise)
- `< -1`: Process group -pid'ye

Dönüş: 0 başarı, -1 hata

`exit`

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

Parametreler:

- `status`: Çıkış kodu (0 başarı, diğerleri hata)

İşlevi:

- Process'i sonlandırır
- atexit() fonksiyonlarını çalıştırır
- Dosyaları kapatır, bellek temizler
- Parent'a signal gönderir

Dizin İşlemleri

`getcwd`

```c
char *getcwd(char *buf, size_t size);
```

Parametreler:

- `buf`: Sonucun saklanacağı buffer (NULL ise malloc yapar)
- `size`: Buffer boyutu

İşlevi:

- Mevcut çalışma dizinini döner
- PWD environment variable'ına benzer

Dönüş: Dizin yolu string'i, hata durumunda NULL

`chdir`

```c
int chdir(const char *path);
```

Parametreler:

- `path`: Geçilecek dizin yolu

İşlevi:

- Process'in çalışma dizinini değiştirir
- "cd" komutunun sistem karşılığı

Dönüş: 0 başarı, -1 hata

`stat` / `lstat` / `fstat`

```c
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Parametreler:

- `pathname`: Dosya yolu (stat/lstat için)

- `fd`: Dosya tanımlayıcısı (fstat için)
- `statbuf`: Dosya bilgilerinin saklanacağı yapı

Farkları:

- `stat`: Symbolic link'i takip eder
- `lstat`: Symbolic link'in kendisi hakkında bilgi
- `fstat`: Açık dosya tanımlayıcısı için

struct stat önemli alanları:

- `st_mode`: Dosya türü ve izinleri
- `st_size`: Dosya boyutu
- `st_mtime`: Son değişiklik zamanı
- `st_uid/st_gid`: Sahip kullanıcı/grup ID

`unlink`

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

Parametreler:

- `pathname`: Silinecek dosya yolu

İşlevi:

- Dosya sisteminden link'i kaldırır
- Link count 0 olunca dosya gerçekten silinir
- Dizinler için kullanılamaz (rmdir gerekli)

Dönüş: 0 başarı, -1 hata

`execve`

```c
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

Parametreler:

- `pathname`: Çalıştırılacak program yolu
- `argv`: Komut satırı argümanları (NULL ile biter)

- `envp`: Environment değişkenleri (NULL ile biter)

İşlevi:

- Mevcut process'i yeni programla değiştirir
- Başarılı olursa geri dönmez
- fork() ile birlikte yeni process oluşturmak için kullanılır

argv[0]: Genelde program adı **envp formatı**: "VAR=value" şeklinde string'ler

Dosya Tanımlayıcı İşlemleri

`dup`

c

```c
int dup(int oldfd);
```

Parametreler:

- `oldfd`: Kopyalanacak dosya tanımlayıcısı

İşlevi:

- Mevcut fd'nin kopyasını oluşturur
- En küçük kullanılabilir fd numarasını kullanır
- Aynı dosyayı işaret eden iki fd oluşur

Dönüş: Yeni fd numarası, hata durumunda -1

`dup2`

c

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

Parametreler:

- `oldfd`: Kaynak dosya tanımlayıcısı
- `newfd`: Hedef fd numarası

İşlevi:

- oldfd'yi newfd'ye kopyalar
- newfd zaten açıksa önce kapatır
- I/O redirection için kullanılır

Kullanım: stdout'u dosyaya yönlendirme

```c
dup2(file_fd, STDOUT_FILENO); // stdout artık dosyaya gider
```

`pipe`

```c
int pipe(int pipefd[2]);
```

Parametreler:

- `pipefd`: 2 elemanlı fd array'i

İşlevi:

- İki process arası iletişim kanalı oluşturur
- pipefd[0]: okuma ucu
- pipefd[1]: yazma ucu
- Shell'de | operatörünün temeli

Kullanım:

```c
int pipefd[2];
pipe(pipefd);
// Child: pipefd[1]'e yaz
// Parent: pipefd[0]'dan oku
```

Dizin İçeriği İşlemleri

`opendir`

```c
DIR *opendir(const char *name);
```

Parametreler:

- `name`: Açılacak dizin yolu

İşlevi:

- Dizini okumak için açar
- DIR stream'i döner
- "ls" komutunun başlangıcı

Dönüş: DIR pointer, hata durumunda NULL

`readdir`

```c
struct dirent *readdir(DIR *dirp);
```

Parametreler:

- `dirp`: opendir'den dönen DIR pointer

İşlevi:

- Dizinden bir sonraki entry'yi okur
- Her çağrıda farklı dosya/dizin döner
- Sıra garanti edilmez

struct dirent alanları:

- `d_name`: Dosya/dizin adı
- `d_type`: Tür (DT_REG, DT_DIR vb.)

Dönüş: dirent pointer, son entry'de NULL

`closedir`

```c
int closedir(DIR *dirp);
```

Parametreler:

- `dirp`: Kapatılacak DIR pointer

İşlevi:

- Dizin stream'ini kapatır
- Kaynakları serbest bırakır

Dönüş: 0 başarı, -1 hata

Hata İşleme

`strerror`

```c
char *strerror(int errnum);
```

Parametreler:

- `errnum`: Hata numarası (genelde errno değeri)

İşlevi:

- Hata numarasını açıklama metnine çevirir
- Thread-safe değil (strerror_r kullan)

Örnek: "No such file or directory"

`perror`

```c
void perror(const char *s);
```

Parametreler:

- `s`: Hata mesajının başına eklenecek string

İşlevi:

- errno'yu otomatik olarak açıklar
- stderr'a yazdırır
- Format: "s: error_message"

Kullanım:

```c
if (open("file.txt", O_RDONLY) == -1) {
    perror("open failed");
}
```

Terminal İşlemleri

`isatty`

```c
int isatty(int fd);
```

Parametreler:

- `fd`: Kontrol edilecek dosya tanımlayıcısı

İşlevi:

- fd'nin bir terminal olup olmadığını kontrol eder
- Interactive/non-interactive ayırımı için

Dönüş: 1 terminal ise, 0 değilse

`ttyname`

```c
char *ttyname(int fd);
```

Parametreler:

- `fd`: Terminal dosya tanımlayıcısı

İşlevi:

- Terminal cihazının adını döner
- Örnek: "/dev/pts/0"

Dönüş: Terminal adı, hata durumunda NULL

`ttyslot`

```c
int ttyslot(void);
```

Parametreler: Yok

İşlevi:

- Mevcut terminal'in slot numarasını döner
- Eski sistem, artık çok kullanılmaz

`ioctl`

c

```c
int ioctl(int fd, unsigned long request, ...);
```

Parametreler:

- `fd`: Cihaz dosya tanımlayıcısı
- `request`: İşlem kodu
- `...`: İşleme özgü argümanlar

İşlevi:

- Cihaz-specific control işlemleri
- Terminal boyutu alma, özel cihaz ayarları

Örnek: Terminal boyutu alma

c

```c
struct winsize ws;
ioctl(STDOUT_FILENO, TIOCGWINSZ, &ws);
```

`getenv`

c

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

Parametreler:

- `name`: Environment değişkeni adı

İşlevi:

- Ortam değişkeninin değerini döner
- Shell değişkenlerine erişim

Örnek:

c

```c
char *path = getenv("PATH");
char *home = getenv("HOME");
```

Dönüş: Değer string'i, bulunamazsa NULL

Terminal Kontrol (termios)

`tcsetattr`

```c
int tcsetattr(int fd, int optional_actions, const struct termios *termios_p);
```

Parametreler:

- `fd`: Terminal fd
- `optional_actions`: Ne zaman uygulanacağı (TCSANOW, TCSADRAIN, TCSAFLUSH)
- `termios_p`: Yeni terminal ayarları

İşlevi:

- Terminal parametrelerini ayarlar
- Raw mode, canonical mode geçişleri

`tcgetattr`

```c
int tcgetattr(int fd, struct termios *termios_p);
```

Parametreler:

- `fd`: Terminal fd
- `termios_p`: Mevcut ayarların saklanacağı yapı

İşlevi:

- Mevcut terminal ayarlarını okur
- Değişiklik öncesi backup almak için

Termcap/Terminfo Fonksiyonları

`tgetent`

```c
int tgetent(char *bp, const char *name);
```

Parametreler:

- `bp`: Terminal bilgi buffer'ı

- `name`: Terminal türü (TERM environment variable)

İşlevi:

- Terminal veritabanından bilgi yükler
- Terminal capabilities'ini hazırlar

`tgetflag`

```c
int tgetflag(const char *id);
```

Parametreler:

- `id`: Boolean capability adı (2 karakter)

İşlevi:

- Terminal'in bir özelliği destekleyip desteklemediğini kontrol eder

`tgetnum`

```c
int tgetnum(const char *id);
```

Parametreler:

- `id`: Numeric capability adı

İşlevi:

- Terminal'in sayısal özelliklerini döner
- Örnek: satır sayısı, sütun sayısı

`tgetstr`

```c
char *tgetstr(const char *id, char **area);
```

Parametreler:

- `id`: String capability adı
- `area`: String'in saklanacağı alan

İşlevi:

- Terminal control string'lerini döner
- Cursor hareketi, renk kodları vb.

`tgoto`

```c
char *tgoto(const char *cap, int col, int row);
```

Parametreler:

- `cap`: Cursor motion capability string'i
- `col`: Sütun numarası
- `row`: Satır numarası

İşlevi:

- Cursor'u belirli pozisyona götürmek için escape sequence oluşturur

`tputs`

```c
int tputs(const char *str, int affcnt, int (*putc)(int));
```

Parametreler:

- `str`: Çıktılanacak terminal control string'i
- `affcnt`: Etkilenen satır sayısı (padding hesabı için)
- `putc`: Karakter çıktı fonksiyonu

İşlevi:

- Terminal control kodlarını doğru timing ile çıktılar
- Eski terminal'lerde gerekli delay'leri ekler

Kullanım Örnekleri

Basit Shell Döngüsü

c

```c
char *line;
while ((line = readline("minishell$ "))) {
    if (*line) add_history(line);
    // Komutu işle
    free(line);
}
```

Process Oluşturma ve Bekleme

c

```c
pid_t pid = fork();
if (pid == 0) {
    // Child process
    execve("/bin/ls", argv, envp);
} else if (pid > 0) {
    // Parent process
    waitpid(pid, &status, 0);
}
```

Pipe ile İletişim

c

```c
int pipefd[2];
pipe(pipefd);
if (fork() == 0) {
    close(pipefd[0]); // Child sadece yazar
    dup2(pipefd[1], STDOUT_FILENO);
    execve("/bin/cat", ...);
} else {
    close(pipefd[1]); // Parent sadece okur
    char buffer[1024];
    read(pipefd[0], buffer, sizeof(buffer));
}
```