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