Task Series 4

4.1 Coroutines

Implement support for concurrent code execution with coroutines (coroutines) where switching is done with saying way. Your implementation must provide the application with an appropriate programming interface, e.g.:

int myroutines_init(co_t *main);	Initialize main coroutine.
int myroutines_create(co_t *co, void (body)(void *), void *arg);	Create a new coroutine.
int myroutines_switchto(co_t *co);	Switch to another coroutine.
int myroutines_destroy(co_t *co);	Coroutine crash.

Base your implementation on the getcontext(), makecontext(), setcontext(), swapcontext() functions that provide the functionality for creating and switching between separate execution contexts.

Test your implementation through a program that creates two coroutines that transfer a file's data via a buffer (task 1.1). The switch from the producer coroutine to the consumer coroutine must be done every time the store is completely full, and the switch from the consumer coroutine to the producer coroutine must be done every time the store is completely empty. When the data transfer is complete, control must return to the main program, which destroys the coroutines, checks that the data transfer was successful (diff between the file and the copy created by the above transfer), and exits.

4.2 Threads with automatic switching

Extend/modify the above implementation to support concurrent code execution with (your) threads , where the switch is done with automatic way. Your implementation must provide an appropriate interface, e.g.:

int mythreads_init();	Environment initialization.
int mythreads_create(thr_t *thr, void (body)(void *), void *arg);	Create a new thread.
int mythreads_yield();	Voluntary rotation.
int mythreads_join(thr_t *thr);	Waiting for thread to terminate.
int mythreads_destroy(thr_t *thr);	Thread destruction.
int mythreads_sem_create(sem_t *s, int val);	Create and initialize a semaphore
int mythreads_sem_down(sem_t *s);	Traffic light reduction
int mythreads_sem_up(sem_t *s);	Raising a traffic light
int mythreads_sem_destroy(sem_t *s);	Destruction of a traffic light

Automatic switching must be implemented through a periodic alarm/timer with appropriate handling of the corresponding signal. The selection of the next thread to execute must be done with the round-robin policy.

To synchronize between threads, implement basic functions binaries traffic lights. All functions (except init) should be safe to run concurrently, properly controlling switching (alarm/timer handling) so as to avoid race conditions.

4.3 Readers/Subscribers

Test the above implementation by developing a solution to the readers/writers problem using your own threads/semaphores, along with a suitable simulation program in the spirit of tasks 2.3/2.4.

Optionally: Extend your implementation so that your own threads run on top of N pthreads system threads (N is given as the runtime argument). Your implementation should distribute the execution of application threads among system threads (either "statically" based on an initial assignment of an application thread when it is created to a system thread, or "dynamically" with the goal that at all times there is a balanced distribution of application threads to system threads).

Tradition: SaturdayJanuary 27, 2024, 11:59 p.m