REAL TIME OPERATING SYSTEM PROGRAMMING-II: Windows CE, OSEK and Real time Linux

Lesson-12:
Real Time Linux

1. Real Time Linux

Linux 2.6.x

- Linux is after Linus Torvalds, father of the Linux operating system
- Linux 2.6.24 latest version of real time Linux (January 2008)
- Linux 2.6.x provides functions for preemptive scheduling
- High resolution timers
- Preemptive interrupt service handler threads

Linux 2.6.x

- All scheduler functions as O(1), which means size of input to a function is linearly correlated with function run time
- Task scheduler support spin lock
- 2^{30} process IDs
- 4095 device types

Linux for Embedded Systems

- Process Management functions.
- Memory Management functions. [For example, allocation, de-allocation, pointers, creating and deleting the tasks.
- File System Functions.

Linux for Embedded Systems

- Shared Memory functions
- Networking System Functions
- Device Control Functions for any peripheral present into a system (computer)
- System call functions are also like an IO device open ( ), close ( ), write ( ) and read ( ) functions.

2. Processes in Linux

Processes in Linux

- Linux uses POSIX processes and threads.
- Linux header files Linux/ types.h and Linux/ shm.h, if included support the system programming for forking processes and shared memory functions in the kernel.
- For shared memory functions the POSIX map are used in Linux.

Processes in Linux…

- A process always creates as child process in the process that uses fork ( ) function.
- A child creates as the copy of the parent with a new process ID
- fork () returns a different process structure *pid* for the parent and *pid* number of child becomes = 0.

Processes in Linux…

- The child process is made to perform different functions than the parent by overloading function of the parent process using execv ( ) function.

Processes in Linux…

- execv ( ) has two arguments, the *function_name* for the child function and *function_arguments* for the child functions.
- Each *process* has its own memory and cannot directly call another *process*. There is no lightweight process as in UNIX

3. Linux Devices

char device

- Character and block devices.
- char device— parallel port, LCD matrix display, or serial port or keypad or mouse.
- Character access — byte-by-byte and analogous to the access from and to a printer device.

Block device

- block device— a file system (disk).
- Linux permits a block device to read and write byte-by-byte like a char device or read and write block-wise like a block device. A part of the block can be accessed

net device

- A net device is a device that handles network interface device (card or adapter) using a line protocols, for example tty or PPP or SLIP. A *network interface* receives or sends packets using a protocol and sockets, and the kernel uses the modules related to packet transmission.

Input, and media devices

- An input device is a device that handles inputs from a device, for example keyboard. An *input* device driver has functions for the standard input devices.
- The *media* device drivers have functions for the voice and video input devices. Examples are video-frame grabber device, teletext device, radio-device (actually a streaming voice, music or speech device).

video device

- A video device is a device that handles the frame buffer from the system to other systems as a char device does or UDP network packet sending device
- The video drivers for the standard video output devices. does. A sound device driver has functions for the standard audio devices. A sound device is a device that handles audio in standard format.

4. Registering and De-registering and related functions of Linux Modules

module initialization

- module initialization, handling the errors, prevention of unauthorized port accesses, usage-counts, root level security and clean up.
- A module creates by compiling without main ( ).
- A module is an object file.
- For example, object module1.o creates from module1.c file by command $ gcc –c {flags} module1.c

init_module()

- Called before the module is inserted into the kernel.
- The function returns 0 if initialization succeeds and –ve value if does not.
- The function registers a handler for something with the kernel.
- Alternatively it replaces one of the kernel functions by overloading.

insmod

- Inserts module into the Linux kernel. The object file module1.o, inserts by command $ insmod module1.o.

rmmod

- A module file module1.o is deleted from the kernel by command $ rmmod module1

cleanup

- A kernel level void function, which performs the action on an rmmod call from the execution of the module. The cleanup_module() is called just before the module is removed. The cleanup_module() function negates whatever init_module() did and the module unloads safely

Registering modules

- register_capability— A kernel level function for registering
- unregister_capability— A kernel level function for deregistering
- register_symtab —A symbol table function support, which exists as an alternative to declaring functions and variables static

5. IPC functions

IPC functions

- Linux/ipc.h included to support IPCs
- signals on an event─ Linux header file Linux/signal.h, included to support
- multithreading ─ Linux/pthread.h included
- mutex and semaphores ─ Linux/sem.h included
- Message queues ─ Linux/msg.h

Linux functions for the thread properties and signal

Thread properties	Threads of same process share the memory space and manage access permissions. The IPCs are used for synchronization objects (interprocess communication objects) and threads.
Signal functions	1. struct sigaction signal_1; /* statement in C defines a structure signal_1. */ 2. signal_1.sa_flags =0; /* Sets flags = 0*/ 3. signal_1.*sa_handler* = handlingfunction_1; /* Defines signal handler as a user defined function handlefunction_1. */ 4. sigemptyset (&signal_1.sa_mask); /* Defines signal as empty and sa_mask masks the execution of signal handler. */ 5. sigaction (SIGINIT, &signal_1, 0); /* Initiates signal function handlingfunction_1() using the structure signal_1. */

Linux functions for the POSIX threads

Thread functions	1. struct pthread_t clientThd_1, clientThd_2, ServingThd; /* statement in C defines three structures for POSIX threads clientThd_1, clientThd_2, ServingThd. */ 2. pthread_create (&clientThd_1, NULL, 0; thread_1, NULL) /* creates thread with structure clientThd_1 and calling function thread_1. NULL is the parameter passed. Similarly the threads clientThd_2, ServingThd and others can be created*/ The function returns an integer *createdstate* which is not 0 in case thread creation fails. 3. pthread_self () ; /* Returns (gets) ID of the function pthread_self calling thread (self) */ 4. pthread_join (&clientThd_1, &threadNew) /* joins thread with structure clientThd_1 and threadNew. */ The function returns an integer *createdstate* which is not 0 in case thread creation fails. 5. pthread_exit ("text") /* exits the thread under execution and text displays on console on exit. */ 6. sleep (*sleeptime*); /* The thread sleeps for a period = *sleeptime* nanoseconds. Other thread gets the CPU in sleep interval*/ 7. pthread_kill (); /* Sends 'kill' signal to the thread */

Linux functions for the semaphore IPCs

Semaphore functions	1. struct sem_t sem1, sem2; /* statement in C defines three structures for semaphore structure (event control block), sem1 and sem2. */ 2. sem_init (); The arguments are semaphore to be initiated and *options*. The function returns an integer *createdstate* which is not 0 in case semaphore initiation fails 3. sem_wait (&sem1); /* wait for the semaphore sem1 */ 4. sem_post (&sem1); /* post the semaphore sem1 */ 5. sem_destroy (&sem1); /* destroy the semaphore sem1 */
Mutex functions	1. struct pthread_mutex_t ms1, ms2; /* statement in C defines three structures for semaphore structure (event control block), ms1 and ms2. */ 2. pthread_mutex_init (); Initiates a mutex. The arguments are semaphore to be initiated (ms1 or ms2 or other) and parameter *options*. The function returns an integer *createdstate* which is not 0 in case mutex initiation fails 3. pthread_mutex_lock (); The argument is mutex ms1 or ms2 or other*/ 4. pthread_mutex_unlock (); The argument is mutex ms1 or ms2 or other*/ 5. pthread_mutex_destroy (); The argument is mutex ms1 or ms2 or other*/

Linux functions for message-queue IPCs

Feature	*Description*
Message queue functions	1. pthread_mq_open ( ), mq_close ( ) and mq_unlink ( ) initialise, close and remove a named queue. unlink ( ) does not destroy the queue immediately but prevents the other tasks from using the queue. The queue will get destroyed only if the last task closes the queue. Destroy means to de-allocate the memory associated with queue ECB. 2. pthread_mq_setattr ( ) sets the attributes. 3. pthread_mq_lock ( ) and pthread_mq_unlock ( ) unlock and lock a queue. 4. pthread_mq_send ( ) and pthread_mq_receive ( ) to send and receive into a queue. mq_send four arguments are msgid (message queue ID), (void *) &qsendingdata (pointer to address of user data), sizeof (qsendingdata) and 0. mq_ receive five arguments are msgid (message queue ID), (void *) &qreceivedata (pointer to address of bufferdata), sizeof (qreceivingdata), 0 and 0. 5. pthread_mq_notify ( ) signals to a single waiting task that the message is now available. The notice is exclusive for a single task, which has been registered for a notification. (Registered means later on takes note of the pthread_mq_notify.) 6. The function pthread_mq_getattr ( ) retrieves the attribute of a POSIX queue.
Queues of functions	The queue of functions are equivalent to character devices, accessed via POSIX read/write/ open/ioctl system calls.

Linux functions for timers

Time functions	1. nanosleep (sleeptime); /* Sleep for nanosecond period defined in argument sleeptime /* 2. clock_getthrtime ( ) ; /* getthrtime returns high resolution time (in ns) of the clock named clock. The clock is name of the clock (system clock) specified earlier. */ 3. getthrtime ( ) ; /* getthrtime returns high resolution time (in ns) in a thread in which it is used as argument. */ 4. clock_gettime ( ) ; /* gettime returns time of the clock named clock. */ 5. clock_settime ( ) ; /* settime sets time of the clock named clock. */

Linux functions for shared memory

Shared memory functions

Used as an alternative to using the IPCs.

1. include <shm.h> /* Includes shared memory header*/.
2. include "common.h> /* Includes common memory file header*/.
3. struc common_struct sharedblck; /* Specify a new structure sharedblck */.
4. int shmID = shmget ((key_t) num, sizeof (struc common_struct), 0666 | IPC_CREAT); /* shmget ( ) function arguments are (key_t) num (= 2377), common structure size, and a option for shared memory creation or use of IPC. Return ID of shared memory structure*/.
5. sharedMem1 = shmat (shmID, (void *)0, 0) /* Pointer to shared memory block sharedMem1 is found by function shmat. The arguments are ID of shared memory block, null pointer function and option 0. */.
6. int shmem_status = shmctl (shmID, IPC_RMID, 0); /* shmctl ( ) function arguments are The arguments are ID of shared memory block, IPC_RMID pointer and option 0. Return shared memory status shmem_status */
7. int shmemdetect_status = shmdt (sharedMem1); /* shmdt ( ) function argument is pointer to shared memory block sharedMem1. Returns –1 if not detected. */

Summary

We learnt

- Linux 2.6.x is for real time system
- Preemptive scheduler
- Linux 2.6.x provides functions for preemptive scheduling
- High resolution timers
- Preemptive interrupt service handler threads

We learnt

- Linux uses POSIX processes, threads, shared memory, POSIX mmap, and POSIX queues.
- Linux has number of interfaces for user. For example, X-Windows for GUI and csh (for C shell). Linux has number of character, block and network interface devices and has device drivers for user programming environment.

We learnt

- Linux supports a module initialization, handling the errors, prevention of unauthorized port accesses, usage-counts, root level security, insert, remove and cleanup functions

End Lesson-12 on

Real Time Linux