Advanced System Programming

Advanced System Programming

Agenda

- Qemu networking setup
- Thread Synchronization
- Thread Debugging
- Process Scheduling

Building Qemu Kernel Image

- Downloading the Kernel and extract

  www.kernel.org

  # tar -xvzf linux-5.1.16.tar.gz

  # cd linux-5.1.16/

- copy configuration file

  # cp arch/arm/configs/vexpress_defconfig .config

- configure kernel

  # make menuconfig ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-

- Build kernel

  # make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-

Building Qemu Kernel Image

- Build kernel modules

  # make ARCH=arm CROSS_COMPILE=CROSS_COMPILE=arm-linux-gnueabihf- modules

- Setup Network

```
sudo ifconfig tap0 down
sudo ifconfig br0 down
sudo brctl delif br0 ens36
sudo brctl delif br0 tap0
sudo brctl delbr br0
sudo tunctl -d tap0
sudo brctl addbr br0
sudo tunctl -u 1000
sudo ifconfig ens36 0.0.0.0 promisc up
sudo ifconfig tap0 0.0.0.0 promisc up
```

Building Qemu Kernel Image

sudo ifconfig br0 172.16.78.100 netmask 255.255.255.0 up

sudo brctl stp br0 off

sudo brctl setfd br0 1

sudo brctl sethello br0 1

sudo brctl addif br0 ens36

sudo brctl addif br0 tap0

sudo bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'

sudo bash -c 'echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp'

sudo chmod 666 /dev/net/tun

- Boot kernel image with GUI

  # qemu-system-arm -M vexpress-a9 -m 512M -net nic,model=lan9118 -net tap,ifname=tap0 -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs.img.gz -append "root=/dev/ram rdinit=/linuxrc"

Building Qemu Kernel Image

- Boot kernel image without GUI

```
# qemu-system-arm -M vexpress-a9 -m 512M -net nic,model=lan9118 -net tap,ifname=tap0 -dtb linux-5.1.16/arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic -kernel linux-5.1.16/arch/arm/boot/zImage -initrd rootfs.img.gz -append "root=/dev/ram console=ttyAMA0 rdinit=/linuxrc"
```

Debugging – GDB Fork

set follow-fork-mode mode

Set the debugger response to a program call of fork or vfork. A call to fork or vfork creates a new process. The mode argument can be:

parent

The original process is debugged after a fork. The child process runs unimpeded. This is the default.

child

The new process is debugged after a fork. The parent process runs unimpeded.

show follow-fork-mode

Print state of follow-fork-mode

Debugging - GDB

set detach-on-fork mode

Tells gdb whether to detach one of the processes after a fork, or retain debugger control over them both.

on

The child process (or parent process, depending on the value of follow-fork-mode) will be detached and allowed to run independently. This is the default.

off

Both processes will be held under the control of GDB. One process (child or parent, depending on the value of follow-fork-mode) is debugged as usual, while the other is held suspended.

Debugging - GDB

(gdb) info inferior

(gdb) inferior

Debugging – GDB Thread

(gdb) info thread

(gdb) thread

Debugging – GDB coredump

You can also start with both an executable program and a core file specified:

 # generate-core-file [Filename]

Debugging – Thread Sanitizer

ThreadSanitizer is a tool that detects data races. It consists of a compiler instrumentation module and a run-time library. Typical slowdown introduced by ThreadSanitizer is about 5x-15x. Typical memory overhead introduced by ThreadSanitizer is about 5x-10x.

-fsanitize=thread

Debugging – Valgrind

helgrind

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.

The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Debugging – Valgrind

helgrind

Helgrind can detect three classes of errors, which are discussed in detail in the next three sections:

- Misuses of the POSIX pthreads API.
- Potential deadlocks arising from lock ordering problems.
- Data races -- accessing memory without adequate locking or synchronisation.

 # valgrind --tool=helgrind program_name

Synchronization mechanism – Critical Section

Critical Section

- The ultimate cause of most bugs involving threads is that they are accessing the same data at the same time.

- The section of code which is responsible to access the shared data, is called *Critical Section* .

- A critical section is part of code that should be executed completely or not at all (a thread should not be interrupted when it is in this section)

- If you do not protect the *Critical Section*, your program might crash because of *Race Condition*.

Synchronization mechanism – Race Condition

Race Condition

- Race Condition is a condition in which threads are racing each other to change the same data structure.

- Because there is no way to know when the system scheduler will interrupt one thread and execute the other one, the buggy program may crash once and finish regularly next time.

- To eliminate race conditions, you need a way to make operations *atomic* (uninterruptible).

Synchronization mechanism – Threads

- Counting Semaphores
  - Permit a limited number of threads to execute a section of the code
- Binary Semaphores - Mutexes
  - Permit only one thread to execute a section of the code
- Condition Variables
  - Communicate information about the state of shared data

Synchronization mechanism – PosixSemaphore

Data type

Semaphore is a variable of type **sem_t**

Include **<semaphore.h>**

Atomic Operations

```
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);

Initialize an semaphore

Returns

0 on success

-1 on failure, sets **errno**

Parameters

sem:

Target semaphore

pshared:

0: only threads of the creating process can use the semaphore

Non-0: other processes can use the semaphore

value:

Initial value of the semaphore

You cannot make a copy of a semaphore variable!!!

Synchronization mechanism – PosixSemaphore

- Sharing semaphores between threads within a process is easy, use **pshared==0**
- A non-zero **pshared** allows any process that can access the semaphore to use it Places the semaphore in the global (OS) environment Forking a process creates copies of any semaphore it has

Synchronization mechanism – PosixSemaphore

sem_init can fail

On failure

sem_init returns -1 and sets **errno**

errno	cause
EINVAL	**Value > sem_value_max**
ENOSPC	Resources exhausted
EPERM	Insufficient privileges

```
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
  perror("Failed to initialize semaphore semA");
```

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_destroy(sem_t *sem);

Destroy an semaphore

Returns

0 on success

-1 on failure, sets **errno**

Parameters

sem:

Target semaphore

Notes

Can destroy a **sem_t** only once

Destroying a destroyed semaphore gives undefined results

Destroying a semaphore on which a thread is blocked gives undefined results

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_post(sem_t *sem);

Unlock a semaphore - same as signal

Returns

0 on success

-1 on failure, sets **errno** (== **EINVAL** if semaphore doesn't exist)

Parameters

sem:

Target semaphore

sem > 0: no threads were blocked on this semaphore, the semaphore value is incremented

sem == 0: one blocked thread will be allowed to run

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_wait(sem_t *sem);

Lock a semaphore

  Blocks if semaphore value is zero

Returns

  0 on success

  -1 on failure, sets **errno** (== **EINTR** if interrupted by a signal)

Parameters

  sem:

  Target semaphore

  sem > 0: thread acquires lock

  sem == 0: thread blocks

Synchronization mechanism – PosixSemaphore

#include <semaphore.h>

int sem_trywait(sem_t *sem);

Test a semaphore's current condition

  Does not block

Returns

  0 on success

  -1 on failure, sets **errno** (**==** **AGAIN** if semaphore already locked)

Parameters

  sem:

    Target semaphore

    sem > 0: thread acquires lock

    sem == 0: thread returns

Synchronization mechanism – Mutex

A typical sequence in the use of a mutex

1. Create and initialize **mutex**
2. Several threads attempt to lock **mutex**
3. Only one succeeds and now owns **mutex**
4. The owner performs some set of actions
5. The owner unlocks **mutex**
6. Another thread acquires **mutex** and repeats the process
7. Finally **mutex** is destroyed

Synchronization mechanism – Mutex

Creating a mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Initialize a pthread mutex: the mutex is initially unlocked

Returns

- 0 on success
- Error number on failure
  - EAGAIN: The system lacked the necessary resources; ENOMEM: Insufficient memory ; EPERM: Caller does not have privileges; EBUSY: An attempt to re-initialise a mutex; EINVAL: The value specified by attr is invalid

Parameters

- mutex: Target mutex
- attr:
  - NULL: the default mutex attributes are used
  - Non-NULL: initializes with specified attributes

Synchronization mechanism – Mutex

#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);

Destroy a pthread mutex

Returns

0 on success

Error number on failure

EBUSY: An attempt to re-initialise a mutex; EINVAL: The value specified by attr is invalid

Parameters

mutex: Target mutex

Synchronization mechanism – Mutex

Locking/unlocking a mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Returns

0 on success

Error number on failure

EBUSY: already locked; EINVAL: Not an initialised mutex; EDEADLK: The current thread already owns the mutex; EPERM: The current thread does not own the mutex

Synchronization mechanism – Conditional Variable

- Used to communicate information about the state of shared data
  - Execution of code depends on the state of
    - A data structure or
    - Another running thread
- Allows threads to synchronize based upon the actual value of data
- Without condition variables
  - Threads continually poll to check if the condition is met

Synchronization mechanism – Conditional Variable

- Signaling, not mutual exclusion
  - A mutex is needed to synchronize access to the shared data
- Each condition variable is associated with a single mutex
  - Wait atomically unlocks the mutex and blocks the thread
  - Signal awakens a blocked thread

Synchronization mechanism – Conditional Variable

Similar to pthread mutexes

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Synchronization mechanism – Conditional Variable

Waiting

Block on a condition variable.

Called with **mutex** locked by the calling thread

Atomically release **mutex** and cause the calling thread to block on the condition variable

On return, **mutex** is locked again

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);

Synchronization mechanism – Conditional Variable

Signaling

int pthread_cond_signal(pthread_cond_t *cond);

unblocks at least one of the blocked threads

int pthread_cond_broadcast(pthread_cond_t *cond);

unblocks all of the blocked threads

Signals are not saved

Must have a thread waiting for the signal or it will be lost

Process Scheduling

Although Linux is a preemptively multitasked operating system, it also provides a system call that allows processes to explicitly yield execution and instruct the scheduler to select a new process for execution:

```
#include <sched.h>

int sched_yield (void);
```

A call to `sched_yield( )` results in suspension of the currently running process, after which the process scheduler selects a new process to run, in the same manner as if the kernel had itself preempted the currently running process in favor of executing a new process. Note that if no other runnable process exists, which is often the case, the yielding process will immediately resume execution. Because of this uncertainty, coupled with the general belief that there are generally better choices, use of this system call is not common.

Process Scheduling

Linux provides several system calls for retrieving and setting a process' nice value. The simplest is `nice( )`:

```
#include <unistd.h>

int nice (int inc);
```

A successful call to `nice( )` increments a process' nice value by `inc`, and returns the newly updated value. Only a process with the `CAP_SYS_NICE` capability (effectively, processes owned by root) may provide a negative value for `inc`, decreasing its nice value, and thereby increasing its priority. Consequently, nonroot processes may only lower their priorities (by increasing their nice values).

On error, `nice( )` returns `-1`. However, because `nice( )` returns the new nice value, `-1` is also a successful return value. To differentiate between success and failure, you can zero out `errno` before invocation, and subsequently check its value. For example:

Process Scheduling

A preferable solution is to use the `getpriority( )` and `setpriority( )` system calls, which allow more control, but are more complex in operation:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

These calls operate on the process, process group, or user, as specified by `which` and `who`. The value of `which` must be one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, in which case `who` specifies a process ID, process group ID, or user ID, respectively. If `who` is `0`, the call operates on the current process ID, process group ID, or user ID, respectively.

A call to `getpriority( )` returns the highest priority (lowest numerical nice value) of any of the specified processes. A call to `setpriority( )` sets the priority of all specified processes to `prio`. As with `nice( )`, only a process possessing `CAP_SYS_NICE` may raise a process' priority (lower the numerical nice value). Further, only a process with this capability can raise or lower the priority of a process not owned by the invoking user.

Like `nice( )`, `getpriority( )` returns `-1` on error. As this is also a successful return value, programmers should clear `errno` before invocation if they want to handle error conditions. Calls to `setpriority( )` have no such problem; `setpriority( )` always returns `0` on success, and `-1` on error.

Process Scheduling

Processes can manipulate the Linux scheduling policy via `sched_getscheduler( )` and `sched_setscheduler( )`:

```
#include <sched.h>

struct sched_param {
        /* ... */
        int sched_priority;
        /* ... */


};

int sched_getscheduler (pid_t pid);

int sched_setscheduler (pid_t pid,
                        int policy,
                        const struct sched_param *sp);
```

A successful call to `sched_getscheduler( )` returns the scheduling policy of the process represented by `pid`. If `pid` is `0`, the call returns the invoking process' scheduling policy. An integer defined in `<sched.h>` represents the scheduling policy: the first in, first out policy is `SCHED_FIFO`; the round-robin policy is `SCHED_RR`; and the normal policy is `SCHED_OTHER`. On error, the call returns `-1` (which is never a valid scheduling policy), and `errno` is set as appropriate.

Thank you