Signals and other Inter-Process Communication (IPC) Mechanisms

Pradipta De
pradipta.de@sunykorea.ac.kr
Today’s Topic

• Synchronization of user level processes
  – User/User synchronization
  – Kernel/user synchronization

• Different synchronization techniques
  – Signals
  – Pipes and FIFOs
  – Semaphores
  – Shared Memory
What are signals?

- Allow interactions between user mode processes
- Kernel can notify user processes of system level events

Signal: a short message, typically a number indicating the signal, sent to a (group of) processes
- SIGKILL: kill all processes in a group
- SIGSEGV: invalid memory reference by a process

Two purposes:
- Inform process that a specific event has occurred
- Process can execute a signal handler in its code
Signals in Linux

• There are 31 signals handled in Linux
  – Architecture dependent
    • SIGSTOP: stop process execution
  – Some signals are only for specific architectures

• Real-time signals: POSIX standard introduced new signals from 32 to 64
  – Regular signals, sent multiple times, will be delivered once ➔ not queued
  – RT signals are queued, and multiple instances delivered

• There are wrapper functions in user level libraries to deliver signals
  – Kill: send signal to a thread group
  – Sigaction: change action associated with a signal
Signal Mechanism

• Signal sent to a process not executing currently
  – Signal saved by the kernel till the process executes ➔ block a signal till process unblocks

• Two phases
  – Signal generation
    • Update process data structure indicating that signal received
  – Signal delivery
    • Change process execution state to handle the signal

• Properties:
  – Signals may be selectively blocked by a process
  – While executing signal handler, the corresponding signal is masked ➔ cannot be interrupted by another signal of same type, therefore, code is NOT reentrant
Process Response to a Signal

- Ignore the signal
- Default action
  - Terminate
  - Dump
  - Ignore
  - Stop
  - Continue
- Invoke a handler
Data Structures

- Default action type (SIG_DFL)
- SIG_IGN
- Pointer to signal handler

Shared by all processes in thread group
- Use CLONE_SIGHAND
Details of signal handling

• If a process is not running, then kernel updates the PCB (signal related data structures)

• Check for signal pending before resuming user process
  – Check after completing an interrupt or exception handling
  – Calls “do_signal” function ➔ process all non-blocked signals in shared and private signal queue
Executing Signal Handler Code

• Signal handler functions are defined by user processes, and resides in user code segment
  – Prototype declaration
    • void (*signal(int signo, void (*func)(int)))(int);
  – Code sample:
    • if (signal(SIGINT, sig_handler) == SIG_ERR)
      printf("can't catch SIGINT");
  
• *do_signal* calls *handle_signal*
  – *handle_signal* executes in kernel mode, while signal handlers execute in user mode
Executing Signal Handler Code

- Implementation challenge
  - Current process executes signal handler in user mode first before resuming “normal” execution
  - Where will the “hardware context” for normal execution reside if kernel stack is emptied
    - Switch from user to kernel mode empties kernel stack
- Save the hardware context in user mode stack of the current process
- When signal handler terminates, sigreturn system call copies the hardware context back to kernel stack and restores original content of user stack
Executing Signal Handler Code

User Mode

Normal program flow

Signal handler

Return code on the stack

Kernel Mode

do_signal()

handle_signal()

setup_frame()

system_call()

sys_sigreturn()

restore_sigcontext()
Other IPC Mechanisms

- Anonymous PIPEs
- Named PIPEs (FIFOs)
- Message Queues (SysV and POSIX)
- Semaphores (SysV and POSIX)
- FUTEX locks
- Shared memory
- Sockets
Naïve synchronization

- Create temporary files protected by locks to share data
  - Costly since access to filesystem on disk
  - Can we design process communication without interacting with the filesystem
  - Several wrapper functions in libraries
PIPEs

• A one way flow of data between processes
• All data written by a process to a pipe is routed by kernel to another process for reading

• Example: ls | more
  – Same as: ls > temp ; more < temp

• Pipes are open files with no image in the mounted filesystem
  – Pipe call returns two file descriptors
  – int pipe (int pipefd[2]);
    • pipefd[0]: read end of pipe
    • pipefd[1]: write end of pipe
Using pipe

- Example: `ls | more`
  - Invoke pipe syscall ➔ `fd[0]=3, fd[1]=4`
  - Invoke fork twice
  - Invoke close on fd to release fd 3, 4

- First child process:
  - `dup2(4,1)` ➔ write to standard out
  - `close fd 4`
  - `execve (ls)`

- Second child process:
  - `dup2(3,0)` ➔ read from standard in
  - `close fd 3`
  - `execve (more)`

- What if there are multiple processes reading and writing using a pipe?
Pipe Implementation

- Pipe buffer is a page frame that contains data written into but not yet read
- Linux 2.6 implements 16 pipe buffers, organized as a circular buffer
- Implemented as VFS objects with no disk image
- Pipefs filesystem initialized during kernel initialization and has its own functions for handling read/write calls
FIFOs (Named PIPEs)

• How do you open an already existing pipe?
  – Two arbitrary processes cannot share a pipe
• Include a FIFO filename in the system directory tree
  – Create an inode data structure, but rather than owning disk blocks associate with kernel buffer
  – mkfifo system call
    • char *path = "/tmp/fifo";
    • mkfifo(path, 0600);
• IPC Semaphores
  – Counters to provide controlled access to shared data structures
  – Semaphore value is >0 if resource available;
  – Trying to access resource, decrease semaphore value ➔ kernel blocks process till semaphore value is positive
  – On releasing a resource, process increments semaphore value ➔ kernel wakes up waiting process
  – `semget(key_t key, int nsems, int flags)`
    • Key is unique identifier used by different processes to access the resource
    • Nsems is the number of semaphores in this set
    • **Code snippet:**
      ```c
      key = ftok("/home/usr/samplefile", 'E');
      semid = semget(key, 10, 0666 | IPC_CREAT);
      ```
Other IPC

• IPC Shared Memory
  – Access data by placing in common shared memory area
  – Each process adds to its address space a new memory region that maps the page frames associated with the IPC shared memory region
  – **Code snippet:**
    ```c
    key = ftok("/home/USR/file3", 'R');
    shmid = shmget(key, 1024, 0644 |IPC_CREAT);
    data = shmat (shmid, (void *)0, 0);
    ```

• There are limits on IPC shared memory to prevent overuse of memory by user processes
More on IPC Shared Memory
Putting It Together

- Focus on signal mechanism
- Other techniques for IPC

Useful reference: