1 Linux System Calls

- Your program can invoke to perform system-related functions.
- These functions fall into two categories, based on how they are implemented.
  - A *library function* is an ordinary function that resides in a library external to your program.
    - Most of the library functions we've presented so far are in the standard C library, **libc**.
    - A call to a library function is just like any other function call. The arguments are placed in processor registers or onto the stack, and execution is transferred to the start of the function's code, which typically resides in a loaded shared library.
  - A *system call* is implemented in the Linux kernel.
    - When a program makes a system call, the arguments are packaged up and handed to the kernel, which takes over execution of the program until the call completes.
    - A system call isn't an ordinary function call, and a special procedure is required to transfer control to the kernel.
    - Low-level I/O functions such as **open** and **read** are examples of system calls on Linux.
- Note that a library function may invoke one or more other library functions or system calls as part of its implementation.
- Linux currently provides about 200 different system calls. A listing of system calls for your version of the Linux kernel is in **/usr/include/asm/unistd.h**. Some of these are for internal use by the system, and others are used only in implementing specialized library functions.

1.1 Using strace

- Before we start discussing system calls, it will be useful to present a command with which you can learn about and debug system calls.
- The *strace* command traces the execution of another program, listing any system calls the program makes and any signals it receives.

```
$ strace hostname
```

- This produces a couple screens of output. Each line corresponds to a single system call. For each call, the system call's name is listed, followed by its arguments and its return value.
- In the output from *"strace hostname"*, the first line shows the **execve** system call that invokes the hostname program:

```
execve("/bin/hostname",["hostname"], [/* 49 vars */]) = 0
```

- The first argument is the name of the program to run; the second is its argument list, consisting of only a single element; and the third is its environment list, which *strace* omits for brevity.The next 30 or so lines are part of the mechanism that loads the standard C library from a shared library file.
- Toward the end are system calls that actually help do the program's work. The *uname* system call is used to obtain the system's hostname from the kernel,

```
uname({sys="Linux",node="myhostname", ...}) = 0
```

- Finally, the *write* system call produces output.

```
write(1,"myhostname\n",11) = 11
```

1.2 access: Testing File Permissions

- The *access* system call determines whether the calling process has access permission to a file.
- It can check any combination of read, write, and execute permission, and it can also check for a file's existence.
- The *access* call takes two arguments.
- The return value is 0 if the process has all the specified permissions.
- If the file exists but the calling process does not have the specified permissions, access returns 1 and sets errno to EACCES (or EROFS, if write permission was requested for a file on a read-only file system).
- The program shown in Fig. 1 uses access to check for a file's existence and to determine read and write permissions.

```
$ ./check-access /mnt/cdrom/README
```

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  char* path = argv[1];
  int rval;
  /* Check file existence.  */
  rval = access (path, F_OK);
  if (rval == 0)
    printf ("%s exists\n", path);
  else {
    if (errno == ENOENT)
      printf ("%s does not exist\n", path);
    else if (errno == EACCES)
      printf ("%s is not accessible\n", path);
    return 0;
  }
  /* Check read access.  */
  rval = access (path, R_OK);
  if (rval == 0)
    printf ("%s is readable\n", path);
  else
    printf ("%s is not readable (access denied)\n", path);
  /* Check write access.  */
  rval = access (path, W_OK);
  if (rval == 0)
    printf ("%s is writable\n", path);
  else if (errno == EACCES)
    printf ("%s is not writable (access denied)\n", path);
  else if (errno == EROFS)
    printf ("%s is not writable (read-only filesystem)\n", path);
  return 0;
}
```

Figure 1: Check File Access Permissions.

1.3 fcntl: Locks and Other File Operations

- The *fcntl* system call is the access point for several advanced operations on file descriptors.
- The first argument to *fcntl* is an open file descriptor, and the second is a value that indicates which operation is to be performed.
- The *fcntl* system call allows a program to place a read lock or a write lock on a file, somewhat analogous to the mutex locks.
- A read lock is placed on a readable file descriptor, and a write lock is placed on a writable file descriptor.
- More than one process may hold a read lock on the same file at the same time, but only one process may hold a write lock, and the same file may not be both locked for read and locked for write.
- Note that placing a lock does not actually prevent other processes from opening the file, reading from it, or writing to it, unless they acquire locks with *fcntl* as well. The program in Fig. 2 opens a file for writing whose name is provided on the command line, and then places a write lock on it.

  ```
  $ gcc -o lock-file lock-file.c
  $ touch /tmp/test-file
  $ ./lock-file /tmp/test-file
  ```

  Now, in another window, try running it again on the same file.

  ```
  $ ./lock-file /tmp/test-file
  ```

  opening /tmp/test-file

- Linux provides another implementation of file locking with the *flock* call. The *fcntl* version has a major advantage: It works with files on NFS file systems.

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  char* file = argv[1];
  int fd;
  struct flock lock;
  printf ("opening %s\n", file);
  /* Open a file descriptor to the file.  */
  fd = open (file, O_WRONLY);
  printf ("locking\n");
  /* Initialize the flock structure.  */
  memset (&lock, 0, sizeof(lock));
  lock.l_type = F_WRLCK;
  /* Place a write lock on the file.  */
  fcntl (fd, F_SETLKW, &lock);
  printf ("locked; hit enter to unlock... ");
  /* Wait for the user to hit enter.  */
  getchar ();
  printf ("unlocking\n");
  /* Release the lock.  */
  lock.l_type = F_UNLCK;
  fcntl (fd, F_SETLKW, &lock);
  close (fd);
  return 0;
}
```

Figure 2: Create a Write Lock with fcntl.

1.4 fsync: Flushing Disk Buffers

- On most operating systems, when you write to a file, the data is not immediately written to disk. Instead, the operating system caches the written data in a memory buffer, to reduce the number of required disk writes and improve program responsiveness.
- When the buffer fills or some other condition occurs (for instance, enough time elapses), the system writes the cached data to disk all at one time.
- However, this behavior can make programs that depend on the integrity of disk-based records unreliable.
- For example, suppose that you are writing a transaction-processing program that keeps a journal file.
  - The journal file contains records of all transactions that have been processed so that if a system failure occurs, the state of the transaction data can be reconstructed.
  - It is obviously important to preserve the integrity of the journal file whenever a transaction is processed, its journal entry should be sent to the disk drive immediately.
- To help you implement this, Linux provides the *fsync* system call.
- The *fsync* call doesn't return until the data has physically been written.
- The function in Fig. 3 illustrates the use of *fsync*. It writes a single-line entry to a journal file.
- The *fsync* system call enables you to force a buffer write explicitly. You can also open a file for synchronous I/O, which causes all writes to be committed to disk immediately. To do this, specify the **O_SYNC** flag when opening the file with the *open* call.

1.5 getrlimit and setrlimit: Resource Limits

- The *getrlimit* and *setrlimit* system calls allow a process to read and set limits on the system resources that it can consume.
- You may be familiar with the *ulimit* shell command, which enables you to restrict the resource usage of programs you run; these system calls allow a program to do this programmatically.

```
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
const char* journal_filename = "journal.log";
void write_journal_entry (char* entry)
{
int fd = open (journal_filename, O_WRONLY | O_CREAT | O_APPEND, 0660);
  write (fd, entry, strlen (entry));
  write (fd, "\n", 1);
  fsync (fd);
  close (fd);
}
```

Figure 3: Write and Sync a Journal Entry.

- For each resource there are two limits, the **hard** limit and the **soft** limit.
- Both *getrlimit* and *setrlimit* take as arguments a code specifying the resource limit type and a pointer to a *structrlimit* variable.
- The *rlimit* structure has two fields: *rlim_cur* is the soft limit, and *rlim_max* is the hard limit.
- Some of the most useful resource limits that may be changed are listed here, with their codes:
  - *RLIMIT_CPU:* The maximum CPU time, in seconds, used by a program. This is the amount of time that the program is actually executing on the CPU, which is not necessarily the same as wall-clock time. If the program exceeds this time limit, it is terminated with a **SIGXCPU** signal.
  - *RLIMIT_DATA:* The maximum amount of memory that a program can allocate for its data. Additional allocation beyond this limit will fail.
  - *RLIMIT_NPROC:* The maximum number of child processes that can be running for this user. If the process calls fork and too many processes belonging to this user are running on the system, *fork* fails.

```
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>
int main ()
{
  struct rlimit rl;
  /* Obtain the current limits.  */
  getrlimit (RLIMIT_CPU, &rl);
  printf("%d \n",RLIMIT_CPU);
  /* Set a CPU limit of one second.  */
  rl.rlim_cur = 1;
  setrlimit (RLIMIT_CPU, &rl);
  /* Do busy work.  */
  while (1);
  return 0;
}
```

Figure 4: CPU Time Limit Demonstration.

  - *RLIMIT_NOFILE:* The maximum number of file descriptors that the process may have open at one time.

- The program in Fig. 4 illustrates setting the limit on CPU time consumed by a program. It sets a 1-second CPU time limit and then spins in an infinite loop. Linux kills the process soon afterward, when it exceeds 1 second of CPU time.

```
$ ./limit_cpu
CPU time limit exceeded
```

1.6 getrusage: Process Statistics

- The *getrusage* system call retrieves process statistics from the kernel.
  - It can be used to obtain statistics either for the current process by passing *RUSAGE_SELF* as the first argument,
  - or for all terminated child processes that were forked by this process and its children by passing *RUSAGE_CHILDREN*.
- A few of the more interesting fields in **struct rusage** are listed here:

```
#include <stdio.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>
void print_cpu_time()
{
  struct rusage usage;
  getrusage (RUSAGE_SELF, &usage);
  printf ("CPU time: %ld.%06ld sec user, %ld.%06ld sec system\n",
  usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
  usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
}
```

Figure 5: Display Process User and System Times.

  - *ru_utime* A struct timevalue field containing the amount of **user time**, in seconds, that the process has used. User time is CPU time spent executing the user program, rather than in kernel system calls.
  - *ru_stime* A struct timevalue field containing the amount of **system time**, in seconds, that the process has used. System time is the CPU time spent executing system calls on behalf of the process.
  - *ru_maxrss* The largest amount of physical memory occupied by the process's data at one time over the course of its execution.

- The function in Fig. 5 prints out the current process's user and system time.

1.7 The mlock Family: Locking Physical Memory

- The *mlock* family of system calls allows a program to lock some or all of its address space into physical memory. This prevents Linux from paging this memory to swap space, even if the program hasn't accessed it for a while.

- A time-critical program might lock physical memory because the time delay of paging memory out and back may be too long or too unpredictable.

- High-security applications may also want to prevent sensitive data from

being written out to a swap file, where they might be recovered by an intruder after the program terminates.

- Locking a region of memory is as simple as calling *mlock* with a pointer to the start of the region and the region's length.
- For example, to allocate 32MB of address space and lock it into RAM, you would use this code:

```
const int alloc_size = 32 * 1024 * 1024;
char* memory = malloc(alloc_size);
mlock(memory,alloc_size);
```

- To unlock a region, call *munlock*, which takes the same arguments as *mlock*.
- Only processes with superuser privilege may lock memory with *mlock*.
- In the output from the command **top**, the **SIZE** column displays the virtual address space size of each program (the total size of your program's code, data, and stack, some of which may be paged out to swap space). The **RSS** column (for resident set size) shows the size of physical memory that each program currently resides in.
- Include $< sys/mman.h >$ if you use any of the *mlock* system calls.

1.8 mprotect: Setting Memory Permissions

- We showed how to use the *mmap* system call to map a file into memory. Recall that the third argument to *mmap* is a bitwise or of memory protection flags *PROT_READ*, *PROT_WRITE*, and *PROT_EXEC* for read, write, and execute permission, respectively, or *PROT_NONE* for no memory access.
- If a program attempts to perform an operation on a memory location that is not allowed by these permissions, it is terminated with a **SIGSEGV** (segmentation violation) signal.
- After memory has been mapped, these permissions can be modified with the *mprotect* system call.
- Obtaining Page-Aligned Memory; Note that memory regions returned by **malloc** are typically not page-aligned, even if the size of the memory

is a multiple of the page size. If you want to protect memory obtained from **malloc**, you will have to allocate a larger memory region and find a page-aligned region within it.

```
int fd = open ("/dev/zero",O_RDONLY);
char* memory = mmap (NULL, page_size, PROT_READ | PROT_WRITE,
MAP_PRIVATE, fd, 0);
close (fd);
mprotect (memory, page_size, PROT_READ);
```

- An advanced technique to monitor memory access is to protect the region of memory using *mmap* or *mprotect* and then handle the **SIGSEGV** signal that Linux sends to the program when it tries to access that memory.
- The example in Fig. 6 illustrates this technique.

1.9 readlink: Reading Symbolic Links

- The *readlink* system call retrieves the target of a symbolic link.
- Unusually, readlink does not NUL-terminate the target path that it fills into the buffer. It does, however, return the number of characters in the target path, so NUL-terminating the string is simple.
- The small program in Fig. 7 prints the target of the symbolic link specified on its command line.

```
$ ln -s /usr/bin/wc my_link
$ ./print-symlink my_link
```

1.10 sysinfo: Obtaining System Statistics

- The *sysinfo* system call fills a structure with system statistics. Its only argument is a pointer to a **struct sysinfo**.
- Some of the more interesting fields of struct sysinfo that are filled include these:
  - *uptime* Time elapsed since the system booted, in seconds.
  - *totalram* Total available physical RAM
  - *freeram* Free physical RAM

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
static int alloc_size;
static char* memory;
void segv_handler (int signal_number)
{
  printf ("memory accessed!\n");
  mprotect (memory, alloc_size, PROT_READ | PROT_WRITE);
}
int main ()
{
  int fd;
  struct sigaction sa;
/* Install segv_handler as the handler for SIGSEGV.  */
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &segv_handler;
  sigaction (SIGSEGV, &sa, NULL);
/* Allocate one page of memory by mapping /dev/zero.  Map the memory
     as write-only, intially.  */
  alloc_size = getpagesize ();
  fd = open ("/dev/zero", O_RDONLY);
memory = mmap (NULL, alloc_size, PROT_WRITE, MAP_PRIVATE, fd, 0);
  close (fd);
/* Write to the page to obtain a private copy.  */
  memory[0] = 0;
/* Make the the memory unwritable.  */
  mprotect (memory, alloc_size, PROT_NONE);
/* Write to the allocated memory region.  */
  memory[0] = 1;
/* All done; unmap the memory.  */
  printf ("all done\n");
  munmap (memory, alloc_size);
  return 0;
}
```

Figure 6: Detect Memory Access Using mprotect.

```
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
int main (int argc, char* argv[])
{
  char target_path[256];
  char* link_path = argv[1];
/* Attempt to read the target of the symbolic link.  */
  int len = readlink (link_path, target_path, sizeof (target_path) - 1);
  if (len == -1) {
    /* The call failed.  */
    if (errno == EINVAL)
      /* It's not a symbolic link; report that.  */
      fprintf (stderr, "%s is not a symbolic link\n", link_path);
    else
/* Some other problem occurred; print the generic message.  */
      perror ("readlink");
    return 1;
  }
  else {
    /* NUL-terminate the target path.  */
    target_path[len] = '\0';
    /* Print it.  */
    printf ("%s\n", target_path);
    return 0;
  }
}
```

Figure 7: Print the Target of a Symbolic Link.

```
#include <linux/kernel.h>
#include <linux/sys.h>
#include <stdio.h>
#include <sys/sysinfo.h>
int main ()
{
  /* Conversion constants.  */
  const long minute = 60;
  const long hour = minute * 60;
  const long day = hour * 24;
  const double megabyte = 1024 * 1024;
  /* Obtain system statistics.  */
  struct sysinfo si;
  sysinfo (&si);
  /* Summarize intersting values.  */
  printf ("system uptime : %ld days, %ld:%02ld:%02ld\n",
  si.uptime / day, (si.uptime % day) / hour,
  (si.uptime % hour) / minute, si.uptime % minute);
  printf ("total RAM     : %5.1f MB\n", si.totalram / megabyte);
  printf ("free RAM      : %5.1f MB\n", si.freeram / megabyte);
  printf ("process count : %d\n", si.procs);
  return 0;
}
```

Figure 8: Print System Statistics.

  - *procs* Number of processes on the system

- The program in Fig. 8 prints some statistics about the current system.

1.11 uname

- The *uname* system call fills a structure with various system information, including the computer's network name and domain name, and the operating system version it's running.

- The call to uname fills in these fields:

  - *sysname* The name of the operating system (such as Linux).
  - *release, version* The Linux kernel release number and version level.

```
#include <stdio.h>
#include <sys/utsname.h>
int main ()
{
  struct utsname u;
  uname (&u);
printf ("%s release %s (version %s) on %s\n", u.sysname,
          u.release,u.version, u.machine);
  return 0;
}
```

Figure 9: Print Linux Version Number and Hardware Information.

  - *machine* Some information about the hardware platform running Linux. For x86 Linux, this is i386 or i686, depending on the processor.
  - *node* The computer's unqualified hostname.
  - *__domain* The computer's domain name.

- The small program in Fig. 9 prints the Linux release and version number and the hardware information.

2 Inline Assembly Code

- Higher-level languages such as C and C++ run on nearly all architectures and yield higher productivity when writing and maintaining code.
- GNU Compiler Collection permits programmers to add architecture-dependent assembly language instructions to their programs. For example, programs using x86 instructions cannot be compiled on Power PC computers.
- Inline assembly statements permit you to access hardware directly and can also yield faster code.
- An asm instruction allows you to insert assembly instructions into C and C++ programs. For example, this instruction

  ```
  asm ("fsin" :"=t" (answer) : "0" (angle));
  ```

is an x86-specific way of coding this C statement:

```
answer = sin (angle);
```

- The expression sin (angle) is usually implemented as a function call into the math library, but if you specify the $-O1$ or higher optimization flag, GCC is smart enough to replace the function call with a single **fsin** assembly instruction.

2.1 When to Use Assembly Code

- Although **asm** statements can be abused, they allow your programs to access the computer hardware directly, and they can produce programs that execute quickly.
- You can use them when writing operating system code that directly needs to interact with hardware.
- For example, **/usr/include/asm/io.h** contains assembly instructions to access input/output ports directly.
- The Linux source code file **/usr/src/linux/arch/i386/kernel/process.c** provides another example, using **hlt** in idle loop code.
- You should use inline assembly to speed up code only as a last resort. Current compilers are quite sophisticated and know a lot about the details of the processors for which they generate code.
- Unless you understand the instruction set and scheduling attributes of your target processor very well, you're probably better off letting the compiler's optimizers generate assembly code for you for most operations.
- Occasionally, one or two assembly instructions can replace several lines of higher level language code.
- For example, determining the position of the most significant nonzero bit of a nonzero integer using the C programming languages requires a loop or floating-point computations.
- Many architectures, including the **x86**, have a single assembly instruction (**bsr**) to compute this bit position.

2.2 Simple Inline Assembly

- Here we introduce the syntax of **asm** assembler instructions with an x86 example to shift a value 8 bits to the right:

```
asm ("shrl $8, %0" : "=r" (answer) : "r" (operand) : "cc");
```

  - The first section contains an assembler instruction and its operands. In this example, **shrl** right-shifts the bits in its first operand.
    - Its first operand is represented by %0.
    - Its second operand is the immediate constant $8.
  - The second section specifies the outputs.
    - The instruction's one output will be placed in the C variable **answer**, which must be an *lvalue*. The string *"=r"* contains an equals sign indicating an output operand and an *r* indicating that **answer** is stored in a register.
  - The third section specifies the inputs. The C variable operand specifies the value to shift.
    - The string *"r"* indicates that it is stored in a register but omits an equals sign because it is an input operand, not an output operand.
  - The fourth section indicates that the instruction changes the value in the condition code **cc** register.

2.2.1 Converting an asm to Assembly Instructions

- GCC's treatment of **asm** statements is very simple. It produces assembly instructions to deal with the **asm**'s operands, and it replaces the **asm** statement with the instruction that you specify. It does not analyze the instruction in any way.

- For example, GCC converts this program fragment

```
double foo, bar;
asm ("mycool_asm %1, %0" : "=r" (bar) : "r" (foo));
```

  to these x86 assembly instructions:

```
movl -8(%ebp),%edx
movl -4(%ebp),%ecx
#APP
mycool_asm %edx, %edx
#NO_APP
movl %edx,-16(%ebp)
movl %ecx,-12(%ebp)
```

- **foo** and **bar** each require two words of stack storage on a 32-bit x86 architecture.The register **ebp** points to data on the stack.
- The first two instructions copy **foo** into registers *EDX* and *ECX* on which *mycool_asm* operates.
- The compiler decides to use the same registers to store the answer, which is copied into bar by the final two instructions. It chooses appropriate registers, even reusing the same registers, and copies operands to and from the proper locations automatically.

2.3 Example

- The x86 architecture includes instructions that determine the positions of the least significant set bit and the most significant set bit in a word.
- The processor can execute these instructions quite efficiently. In contrast, implementing the same operation in C requires a loop and a bit shift.
- The **bsrl** assembly instruction computes the position of the most significant bit set in its first operand, and places the bit position (counting from 0, the least significant bit) into its second operand.
- To place the bit position for number into position, we could use this **asm** statement:

  ```
  asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
  ```

  One way you could implement the same operation in C is using this loop:

  ```
  long i;
  for (i = (number >> 1), position = 0; i != 0; ++position)
  i >>= 1;
  ```

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char* argv[])
{
  long max = atoi (argv[1]);
  long number;
  long i;
  unsigned position;
  volatile unsigned result;
  /* Repeat the operation for a large number of values.  */
  for (number = 1; number <= max; ++number) {
    /* Repeatedly shift the number to the right, until the result is
       zero.  Keep count of the number of shifts this requires.  */
    for (i = (number >> 1), position = 0; i != 0; ++position)
      i >>= 1;
    /* The position of the most significant set bit is the number of
       shifts we needed after the first one.  */
    result = position;
  }
  return 0;
}
```

Figure 10: Find Bit Position Using a Loop.

- To test the relative speeds of these two versions, we'll place them in a loop that computes the bit positions for a large number of values.
- The program in Fig. 10 does this using the C loop implementation.The program loops over integers, from 1 up to the value specified on the command line. For each value of number, it computes the most significant bit that is set.
- The program in Fig. 11 does the same thing using the inline assembly instruction.
  - Note that in both versions, we assign the computed bit position to a volatile variable result.
  - This is to enforce the compiler's optimizer so that it does not eliminate the entire bit position computation; if the result is not used or stored in memory, the optimizer eliminates the computation as "dead code".

```
$ gcc -O2 -o bit-pos-loop bit-pos-loop.c
$ gcc -O2 -o bit-pos-asm bit-pos-asm.c
```

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char* argv[])
{
  long max = atoi (argv[1]);
  long number;
  unsigned position;
  volatile unsigned result;
  /* Repeat the operation for a large number of values.  */
  for (number = 1; number <= max; ++number) {
    /* Compute the position of the most significant set bit using the
       bsrl assembly instruction.  */
    asm ("bsrl %1, %0" : "=r" (position) : "r" (number));
    result = position;
  }
  return 0;
}
```

Figure 11: Find Bit Position Using bsrl.

```
$ time ./bit-pos-loop 250000000
$ time ./bit-pos-asm 250000000
```

- Notice that the version that uses inline assembly executes a great deal faster.