Lecture 5: System Functionality

Lecturer: Prof. Zichen Xu

Linux Programming Report

- Midterm report shall be a double column no more than four-page report using Latex.
- The topic shall be
  - An interesting problem
  - Pitfall
  - Loophole in the current or previous operating systems
  - And/or their programming languages

‘The Included but Not Limited to’ Topic List

- Linux Operating System
- FreeBSD Operating System
- Programming Language in Systems
- Android Operating System
- Linux System Security
- GNU and License System
- Open-source Wiki System
- Linux in Auto-drives
- Linux Programming in Distributed Environments

An Interesting Problem, Pitfall, or Loophole

- Examples
  - The socket interface
  - Protocol drivers
  - Network device drivers
  - The pluggable authentication modules (PAM) system
  - MIT License
  - GRUB loader
  - SED v.s. AWK
  - Ext3 and journaling

  …

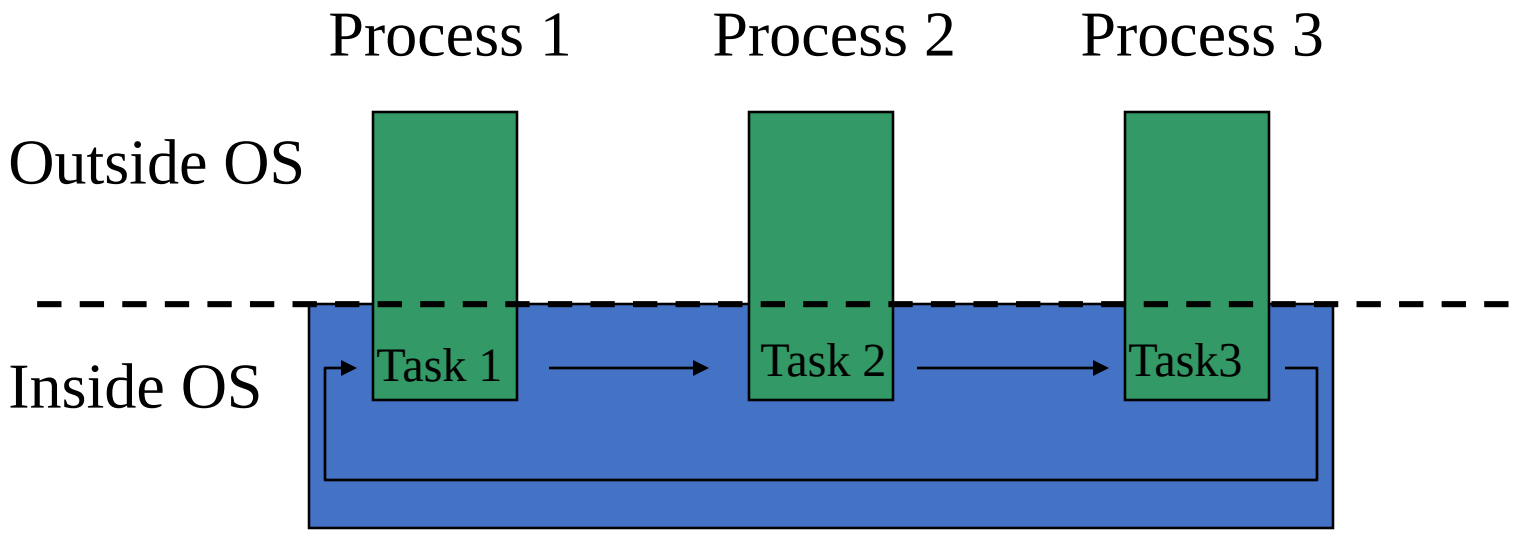
Processes vs. Tasks

- Linux views all work as a series of tasks imposed on the OS.
- Processes and threads are essentially viewed in the same way, except that resources allocated are different

5

Processes vs. Tasks

- Linux views all work as a series of tasks imposed on the OS. Processes and threads are essentially viewed in the same way, except that resources allocated are different



Implementation of Processes and Threads

Categories of information in the process descriptor:

- Scheduling parameters
- Memory image
- Signals
- Machine registers
- System call state
- File descriptor table
- Accounting
- Kernel stack
- Miscellaneous

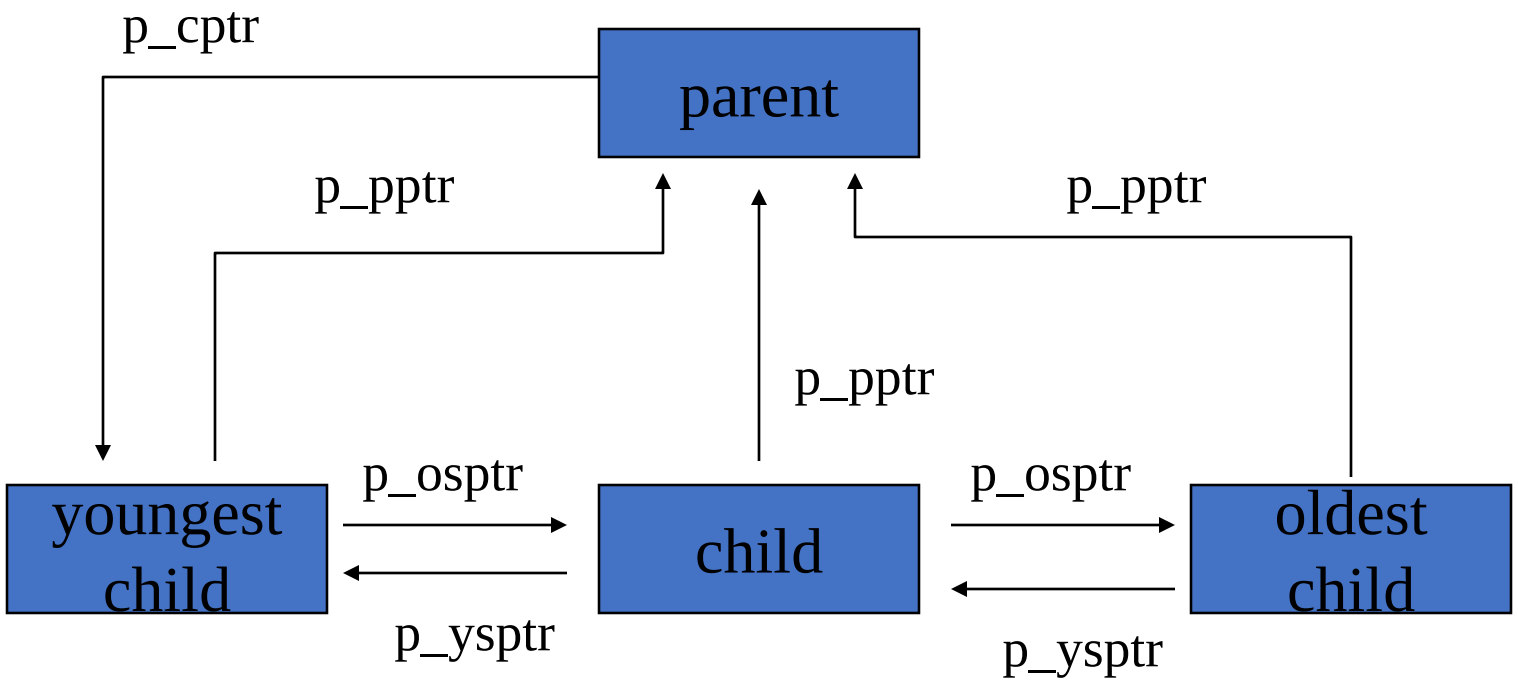
Process Descriptor - task_struct -(see /usr/include/linux/sched.h)

- Individual variables:
  - state
  - priority
  - flags
  - signal
- **Pointers to queues, structs, etc.**
  - **fs_struct**
  - **mm_struct**
  - **task Lists**
  - **“family”**

Process States

- TASK_RUNNING
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_STOPPED
- TASK_ZOMBIE

Process Relationships



pid_t getpid(void);
pid_t getppid(void);

The Process List

- Tasks managed through a doubly linked list. Each task_struct includes:
  - struct task_struct *next_task;
  - struct task_struct *prev_task;
  - List terminated at global variable **init_task**

Process and thread creation

- Process Creation
  - fork ( )
  - execve( )
  - clone( )
- **Thread Creation**
  - clone( )
  - pthread_create( )

Creating Processes - fork( )

- **pid_t fork(void);**

- Creates a child process that differs from the process only in its PID and PPID, and in the fact that resource utilizations are set to 0. Linux uses a copy-on-write procedure.
- **Child process returns PID = 0;**
- **Parent process returns child PID**

Processes in Linux

```
pid = fork( );                          /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
        handle_error( );                /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                        /* parent code goes here. /*/
} else {
                                        /* child code goes here. /*/
}
```

Process creation in Linux.

Process Fork example

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#define LB_SIZE 128

int main ()
{
  time_t now;
  char lineBuf[LB_SIZE];
  pid_t  pid;
  void work (char *name);
  time (&now);
  printf ("Fork test at %s\n", ctime(&now));
```

Process Fork example

```
        if ((pid = fork() ) < 0)   {
        printf("Couldn't fork process!\n");
        exit(1);
  }
  else if (pid == 0) {    // child process
        work ("child");
        return 0;
  }
  work("main");
  printf ("Closing main program...\n");
  return 0;
}
void work(char * name)   {
  printf("This is the %s work cycle\n",name);
  sleep (1);
}
```

Process Fork example (Output)

```
[good]@good308]# gcc -o proc_fork proctest.c
[root]@good308]# ./proc_fork
Fork test at Wed Apr 4 14:57:53 2018

This is the main work cycle
Closing main program...

This is the child work cycle
```

Creating New Processes - execve( )

```
int execve (
    const char *filename,
    char *const argv[ ],
    char *const envp[ ]);
```

filename: name of executable file (or script) to be executed

argv[]: list of arguments to be passed to executable file
(an array of strings. May use NULL)

envp[]: list of environment variables to be used
(an array of strings. May use NULL)

No return on success. -1 on failure

Creating Processes - Execve

- Used to call an executable program from a child process. 6 varieties available, depending on options needed:

	Path-name	file-name	Arg list	argv[ ]	environ	envp[ ]
EXECL	X		X		X	
EXECLP		X	X		X	
EXECLE	X		X			X
EXECV	X			X	X	
EXECVP		X		X	X	
EXECVE	X			X		X

New Process Example (exec.cpp)

```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <errno.h>
extern char **environ;
extern int errno;
pid_t pid;
using namespace std;
```

```
int main (int argc, char* argv[])  {
        FILE *fin;
        char lineBuf[128];
        char *progname= argv[1];
        char myname[15] = "Bob";
        char mynum[5] = "5";
        static int result;
        char *args[4];
```

New Process Example

```
if (argc == 2)
        strcpy (progname, argv[1]);
else    {
        cout << "Usage: " << argv[0] << " progname" << endl;
        exit(1);
}
pid = fork();
if (pid==0)        {  //We are in the child process
        args[0] = progname;
        args[1] = (char *) &myname;
        args[2] = (char *) &mynum;
        args[3] = (char *) 0; // array must be null terminated
        if ( execv (progname, args) == -1)    {
            cout << "ERROR IN EXECVE " << endl;
            perror("execve");
        }
}
```

New Process Example

```
    else  //We are in the parent process
    {
        cout << “Created new process with process ID " << pid << endl;
        sleep(1);
    }
    return 0;
}//main
```

Child Process (hello.cpp)

```
#include <iostream>
using namespace std;

int main(int argc, char * argv[])
{
   int i, mynum;
   if (argc != 3)   {
      cout << "Usage: : " << argv[0] << " NAME NUMBER" << endl;
      return 0;    }
   cout << "Hello, World!" << endl;
   cout << "Passed in " << argc << " arguments" << endl;
   cout << "They are: " << endl;
   mynum = atoi(argv[2]);
   cout << "Program name is " << argv[0] << endl;
   cout << "Name argument is " << argv[1] << endl;
   cout << "The number is " << mynum << endl;
  return 0;
}
```

New Process Example (Output)

```
$ g++ -o exec exec.cpp
$ g++ -o hello hello.cpp
$ ./exec hello

Created new process with process ID 2795
Hello, World!
Passed in 3 arguments
They are:
Program name is hello
Name argument is Bob
The number is 5
```

Process Management System Calls in Linux

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Figure. Some system calls relating to processes.

A Simple Linux Shell

```
while (TRUE) {                                  /* repeat forever /*/
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, params);          /* read input line from keyboard */

        pid = fork( );                          /* fork off a child process */
        if (pid < 0) {
                printf("Unable to fork0);       /* error condition */
                continue;                       /* repeat the loop */
        }

        if (pid != 0) {
                waitpid (–1, &status, 0);       /* parent waits for child */
        } else {
                execve(command, params, 0);     /* child does the work */
        }
}
```

Figure. A highly simplified shell.

Implementation of Exec

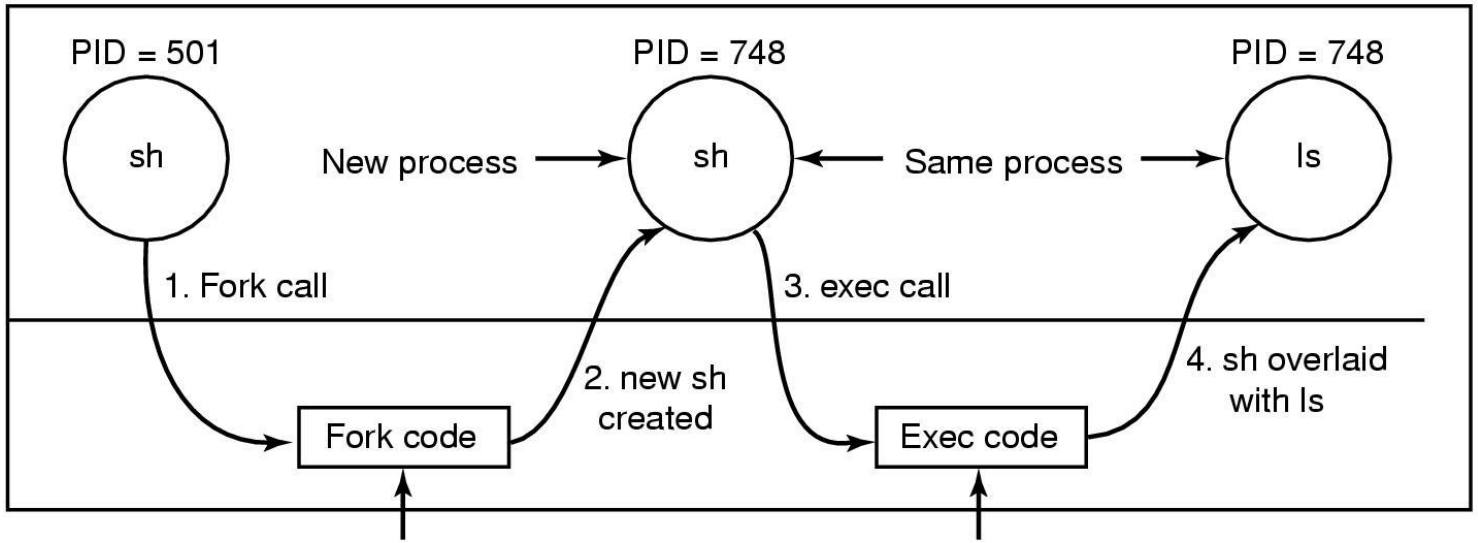


Figure. The steps in executing the command *ls* typed to the shell.

POSIX Threads (1)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure. Some of the Pthreads function calls.

POSIX Threads (2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d0, tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d0, i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d0, status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Figure. An example program using threads.

Creating Threads - _pthread_create()

```
#include <pthread.h>

int pthread_create (
    pthread_t *thread,  //variable to store returned thread ID
    pthread_attr_t *attr,  // thread attributes
    void * (*start_routine)(void *), //function to run
    void * arg);  //args to pass to new thread
```

_pthread_create( ) attributes

- *May be set to **NULL** to use default attributes*
- **detachstate** //control whether the thread is created in the joinable state (default) or not.
- **schedpolicy** //select scheduling policy (default is regular, non-realtime scheduling)
- **schedparam** //set scheduling priority (default is 0)
- **inheritsched** //is sched policy to be inherited by child processes or threads? (default is no)

POSIX Thread example

```
#include <stdio.h>,<sys/time.h>, <sys/types.h>,
  <pthread.h>

#define LB_SIZE 128
int flag;

int main (int argc, char * argv[])  {
  FILE *fin;
  time_t now;
  char lineBuf[LB_SIZE];
  pthread_t tid;  //structure that holds thread number
  int childno, childcount = 3, mainnum = 0;
  void work (void * childnum); //function prototype
```

POSIX Thread example

```
time (&now);
printf ("Thread test at %s\n", ctime(&now));
childno = 1;
while (childno <= childcount) {
   flag = 0;
   if ((pthread_create (&tid,  NULL, (void *)&work,
            (void *)&childno))>0)  {
        printf("Couldn't create new thread!\n");
        exit(1);
   } else //we're in main        {
            while (flag == 0)  ;
            printf("Just created thread %d\n", tid);
            childno++;
   }
}
```

POSIX Thread example

```
  work(&mainnum);
  printf ("Terminating Main program.....\n");
  return 0;
} //end of main

void work(void * arg) {
  int i,j, childnum;
  flag = 1;
  childnum = *(int *) arg;
  for (i = 0; i < 3; i++)        {
     printf("Loop %d of thread %d work cycle\n", i,
  childnum);
     sleep (1);
  }
}
```

POSIX Thread example (Output)

```
# gcc -o thread_test proctest_v4a.c -lpthread
# ./thread_test

Loop 0 of thread 1 work cycle
Just created thread 1026
Loop 0 of thread 2 work cycle
Just created thread 2051
Loop 0 of thread 3 work cycle
Just created thread 3076
Loop 0 of thread 0 work cycle
:
:
Loop 2 of thread 2 work cycle
Loop 2 of thread 3 work cycle
Loop 2 of thread 0 work cycle
Terminating Main program......
```

POSIX Thread example (v2)

```
#include <iostream>,<stdlib.h>,<pthread.h>
#define LB_SIZE 128

using namespace std;
void * work (void * arg); // function prototype

int main (int argc, char * argv[]) {
        time_t now;
        pthread_t tid[3];
        int childno, childcount = 3; index, i, childid[3];
        void * ptReturn; // a pointer to the return value from a thread
```

POSIX Thread example (v2)

```
time (&now);
cout << "Thread Test at " << ctime(&now) << endl;
for (index = 0; index < childcount; index++) {
     childid[index] = index;
     if (( pthread_create(&tid[index], NULL, &work, (void *)&childid[index] )) > 0)
     {
              cout << "Couldn't create new thread!" << endl;
              exit(1);
    } else //we're in main {
              cout << "Thread " << index << " has tid " << tid[index] << endl;
              childno++;
    }
 }
```

POSIX Thread example (v2)

```
cout << "Now, wait for threads to finish ..." << endl;
for (i = 0; i < childcount; i++)
{
    pthread_join (tid[i], &ptReturn);
    cout << "Return value for " << tid[i] << " is: " << (long) ptReturn << endl;
 }
cout << "Terminating Main program ....." << endl;
return 0;
}
```

POSIX Thread example (v2)

```
void * work(void * arg)
{
    int i, childnum; childnum = *(int *) arg; //recast as int pointer, then deref.
    for (i = 0; i < 3; i++)
    {
        cout<< "Loop "<< i << " of thread "<< childnum<<" work cycle"<< endl;
        sleep (1);
     }
     //pthread_exit ((void *) childnum);
     return ((void *)childnum);
}
```

Output for pthread V2

```
$ g++ -o pthread2 pthread_v2.cpp -lpthread
$ ./pthread2

Thread Loop 00 has tid  of thread 0 work cycle
3077553008
Thread 1 has tid 3067063152
Loop 0 of thread 1 work cycle
Thread 2 has tid 3056573296
Now, wait for threads to finish ...
Loop 0 of thread 2 work cycle
Loop 1 of thread 0 work cycle
Loop 1 of thread 1 work cycle
Loop 1 of thread 2 work cycle
Loop 2 of thread 0 work cycle
Loop 2 of thread 1 work cycle
Loop 2 of thread 2 work cycle
Return value for  3077553008 is: 0
Return value for  3067063152 is: 1
Return value for  3056573296 is: 2
Terminating Main program .....
```

UNIX / Linux library inclusion

- Only basic libraries included by default.
- Other libraries (math, pthread, etc.) must be explicitly linked.
- Libraries included in /usr/lib
- File name format is “libxxx.a” where xxx is library name
  - Math library: libm.a use “-lm”
  - POSIX threads library: libpthread.a use “-lpthread”

Creating “Native Threads” clone( )

```
#include <sched.h>

pid = __clone(
  int (*fn) (void *arg), //pointer to function
  void *child_stack, //pointer to a new stack
  int flags, //various flags (see next slide)
  void *arg) //argument passed to function
```

A new stack must be created for the cloned task (process or thread).

The Clone System Call

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PID	New thread gets old PID	New thread gets own PID
CLONE_PARENT	New thread has same parent as caller	New thread's parent is caller

Figure. Bits in the sharing_flags bitmap.

Clone Example

```
#include <stdio.h>, <math.h>, <sys/time.h>, <sys/types.h>, <sched.h>
#define LB_SIZE 128
int flag;
extern int errno;

int main (int argc, char * argv[])
{
  time_t now;
  char lineBuf[LB_SIZE];
  pid_t  pid;
  int childno = 1, mainnum = 0;
    void *csp, *tcsp;
  void test (void *childnum);  //function prototype
```

Clone Example

```
csp=(int *)malloc(8192*8); //create a new stack
  if(csp)
      tcsp=csp+8192*8; //move pointer to the end of the stack
  else
      exit(errno);
time (&now);
printf ("Clone Test run on %s\n\n", ctime(&now));
flag = 0;
childno = 1;
if (( pid = __clone( (void *)&test, tcsp, CLONE_VM,
            (void *)&childno )) < 0)
{
     printf("Couldn't create new thread!\n");
     exit(1);
}
```

Clone Example

```
  else { //we're in main
       while (flag == 0)     ;
       printf("Just created thread %d\n", pid);
  }
  test(&mainnum);
  printf ("Main program is now shutting down\n\n");
  return 0;
}
void* test(void * arg)   {
  int  childnum;
  flag = 1;
  childnum = *(int *) arg;
  printf("Thread %d work cycle\n",childnum);
  sleep (3);
}
```

Clone Example (Output)

```
[root]@bobs-RHLINUX cs431_linux]# gcc -o clone_test proctest_v6.c
[root]@bobs-RHLINUX cs431_linux]# ./clone_test
Clone Test run on Fri Jan 12 14:49:30 2001


Thread 1 work cycle
Just created thread 430
Thread 0 work cycle
Main program is now shutting down
```

Conclusion

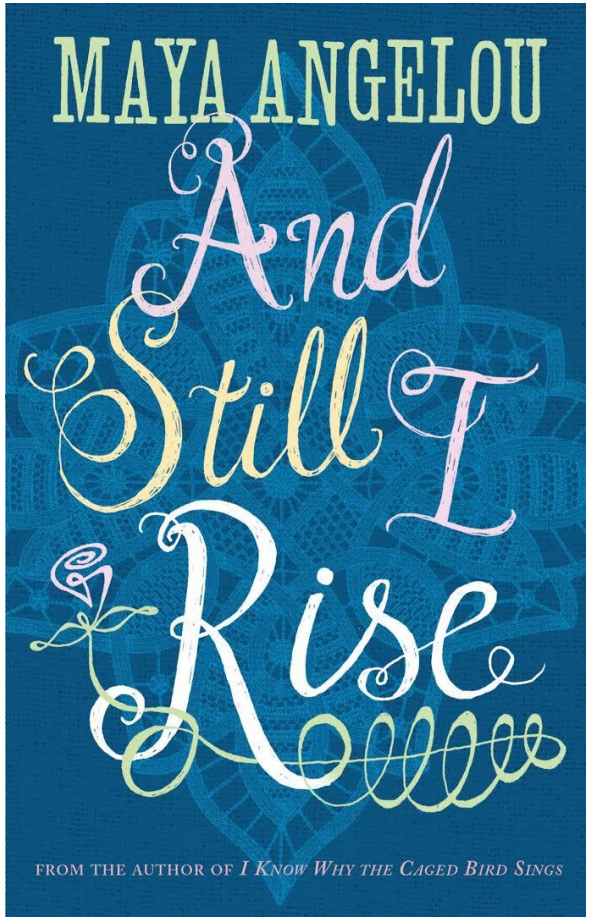
- We have started on system functionality, with a focus on multi-programming
- We have discussed some real code of multi-processing in C coding
- We will start on Linux Programming in C next week
- Reading Assignment: Chapter 2&3 in your textbook

And Still I Rise

Just like moons and like suns,
With the certainty of tides,
Just like hopes springing high,
Still I'll rise.

Did you want to see me broken?
Bowed head and lowered eyes?
Shoulders falling down like teardrops,
Weakened by my soulful cries?

Does my haughtiness offend you?
Don't you take it awful hard
’Cause I laugh like I've got gold mines
Diggin’ in my own backyard.



Kernel Modules

- The provision of kernel modules allows code to be introduced into a running kernel.
- This requires the kernel to be built with this capability, it also requires the commands
  - Insmod and rmmod (plus lsmod, depmod and modprobe)
- Modules can be loaded on demand automatically.

Module programming

- The 2.6 kernel has changed the way module programming is handled.
  - We will look at this later on – for the moment we will deal with 2.4
- Modules under 2.4 are just ordinary unlinked object files (cc –o)
  - Although they must link with the kernel and can bring it down, so they are rather special.

Module programs

- Requires header files
  - These will include others
- Needs an init_module and cleanup_module function
- The return value is important
  - Only return 0.
- Use of printk

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
        printk("<1>Hello world 1.\n");
        return 0;
}

void cleanup_module(void)
{
        printk("KERN_ALERT "Goodbye cruel world 1.\n");
}
```

Using macros

- Use of init & exit macros
- Use of __init and __initdata

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int hello_data __init_data= 47;
static int __init hello2_init(void)
{
        printk("<1>Hello world 2. %d \n", hello_data);
        return 0;
}

static void __exit hello2_exit(void)
{
        printk("KERN_ALERT "Goodbye cruel world 2.\n");
}

module_init(hello2_init);
module_exit(hello2_exit);
```

Module Makefile

```
WARN := −W −Wall −Wstrict−prototypes −Wmissing−prototypes
INCLUDE := −isystem /lib/modules/`uname −r`/build/include
CFLAGS := −O2 −DMODULE −D__KERNEL__ ${WARN} ${INCLUDE}
CC := gcc−3.0
OBJS := ${patsubst %.c, %.o, ${wildcard *.c}}
all: ${OBJS}
.PHONY: clean
clean:
rm −rf *.o
```

Kernel debugging

- Sources:
  - Love 2e: Chapter 18 "Debugging" (2.6)
  - Corbet, Rubini, Kroah-Hartman 3e: Chapter 4 "Debugging Techniques" (2.6)
- Debugging is hard
  - Kernel debugging is harder!
  - Still, many similarities to other large-scale projects
- Need a reproducible bug
  - Intermittent or timing-related bugs very difficult

Types of Kernel Bugs

- Incorrect behaviors
- Corrupt data
- Synchronization errors, races, timing errors
- Performance bugs

Debugging Techniques

- printk()
- Oops
- CONFIG_DEBUG_KERNEL
- SysRq keys
- (Unofficial) kernel debuggers [Not ARM]
  - gdb, kdb, kgdb, nlkd
- /proc
- strace
- User Mode Linux (UML)
- Linux Trace Toolkit (LTT)
- Dynamic Probes (DProbes) [Intel]

printk()

- Very robust! Callable almost anywhere…
- Except very early in boot sequence
- early_printk()
- Circular log buffer
  - klogd (user space) reads /proc/kmsg
  - sends (via syslogd) to /var/log/messages
  - read with dmesg
- Log-levels (message priorities) 0 (high) .. 7 (low)
  - /proc/sys/kernel/printk (threshold)
  - KERN_EMERG, _ALERT, _CRIT, _ERR, _WARNING, _NOTICE, _INFO, _DEBUG

Oops

- Kernel exception handler
  - Kills offending process
  - Prints registers, stack trace with symbolic info
- Some exceptions non-recoverable (panic())
  - Prints message on console, halts kernel
  - Oops in interrupt handler, idle (0) or init (1)
- Oops generated by macros:
  - BUG(), BUG_ON(condition)

ksymoops

- Oops must be "decoded"
  - Associate symbolic names with addresses
- Address info lives in System.map
  - Kernel symbol table generated during compile
  - Module symbols included as well
- ksymoops – user-mode program (file access)
- kallsyms
  - 2.6 technique
  - Reads System.map into kernel memory at init

CONFIG_DEBUG_KERNEL

- Many kernel subsystems have extensive debugging that can be compiled in
- Subsystem specific compile-time debugging
  - _SLAB, _PAGEALLOC, _SPINLOCK, _INIT, _STACKOVERFLOW, _ACPI, _DRIVER, _PROFILING
- Info goes to console and /proc

Magic SysRq Keys

- Special console key sequences recognized by kernel
  - Useful for "system hangs"
  - Must be compiled in CONFIG_MAGIC_SYSRQ
- Alt-SysRq-<key>
  - h: help, b: reboot, s: sync, u: unmount all, etc.
- Toggle on/off:
  - /proc/sys/kernel/sysrq
- Possible to activate remotely by writing char to /proc
  - /proc/sysrq-trigger

(Unofficial) Kernel Debuggers

- No official kernel debugger!
  - Linus believes developers should deeply understand code and not rely on the "crutch" of an interactive debugger
  - Many still use debuggers from time to time (including Linus)
- gdb (from user-mode)
  - gdb <kernel image> /proc/kcore
  - Compile with CONFIG_DEBUG_INFO for symbolic info
  - Problem: caches symbol values *sigh*
- kdb (part of kernel) [Intel only]
  - Kernel halted when kdb runs
- kgdb (debug live kernel remotely via serial port)
  - Two separate patches; in flux; originally from SGI
  - Available for many architectures (but not ARM?)
- nlkd (new debugger from Novell)

/proc

- Export kernel info via /proc
- Write your own /proc files
  - Can be read-only or read-write
  - Provide a special read_proc() function
  - Register with create_proc_read_entry()
  - Details in CRK
- Cleaner interface (seq_file) in 2.6
  - Better for files with lots of data
  - Iterator style (get_next)

strace

- View entry/exit of user-mode processes via system call interface
- Good for debugging new system calls

User Mode Linux (UML)

- Linux kernel emulation that runs as a user-mode process!
- Implemented as architecture port (arch/um)
- Easy to debug with gdb
- Slow
- Not useful for debugging hardware interactions (emulated)

Linux Trace Toolkit

- Generic event trace framework for kernel
- Includes timing information
- Slows things down but provides relative timing info
- Kernel tracing infrastructure + user-mode applications for viewing (graphs, etc.)
- Many supported architectures including ARM

Dynamic Probes (DProbes)

- Contributed by IBM for IA32
- Allows placement of "probes" anywhere in kernel
- Probes are code written in a special interpreted language
- Executed when flow-of-control reaches probe
- New probes can be inserted without kernel rebuild or reboot

Some Debugging Tricks

- Code conditional on UID
  - if (current->uid == 7777) { … }
- Rate limiting printk()
  - Record print time
  - Only print again if interval elapsed