System Programming in C

Concurrency

- At hardware level, multiple devices operate at the same time
- CPUs have internal parallelism – multicore, pipelining
- At application level, signal handling, overlapping of I/O and computation, communications, and sharing of resources
- One of the most difficult problems for the programmer to handle
- Simple example

  ```
  a = b = c = d = 1;
  cobegin
      a = b + c;
      c = b + d;
  coend
  ```

- What is the value of variables when the two arithmetic statements are executed concurrently?
- Problem even greater due to the availability of multiprocessor machines at the desktop level where applications need to exploit all the processors to achieve speed
- *Communication*
  - Conveyance of information from one entity to another
  - The other entity may be specified explicitly (broadcast to one) or the message may be transmitted to everyone to be picked up by the relevant entity
- *Concurrency*
  - Sharing of resources in the same time frame
  - Execution of processes is interleaved in time, on the same CPU
  - Concurrent entities may be threads of execution within programs or other abstract objects (such as processes)
- Processes, threads and resource sharing
  - Program
    * Collection of instructions and data kept in ordinary file on disk
    * The file is marked as executable in the i-node
    * File contents are arranged according to rules established by OS
    * Source program, or text file
    * Machine language translation of the source program, or object file
    * Executable program, complete code output by linker/loader, with input from libraries
  - Process
    * Created by kernel as an environment in which a program executes
    * Program in execution
    * Three segments
      1. Instruction segment
      2. User data segment
      3. System data segment
         · Includes attributes such as current directory, open file descriptors, and accumulated CPU times

        · Information stays outside of the process address space
    * Program initializes the first two segments
    * Process may modify both instructions (rarely) and data
    * Process may acquire resources (more memory, open files) not present in the program
  - Process ID
    * Unique integer to identify a process
    * PID 0 – `swapper`
    * PID 1 – `init`
    * PID 2 – `pagedaemon`

- Process groups
  - Represent a *job* abstraction
  - As an example, processes in a pipeline form a group and the shell acts on those as a single entity
  - Process descriptor contains a field called *process group* ID
    * PID of the *group leader*
  - Login session
    * All processes that are descendants of the process that started a working session on a specific terminal
    * All processes in a process group are in the same login session
    * A login session may have several process groups
    * One of the processes is always in the foreground; it has access to the terminal
    * When a background process tries to access the terminal, it receives a `SIGTTIN` or `SIGTTOUT` signal

Multiprogramming and multitasking

- Multiprogramming
  - A mode of operation that provides for the interleaved execution of two or more programs by a single processor
  - Cycle stealing
    * A mechanism by which the OS assigns higher priority to an I/O-bound process compared to a CPU-bound process
    * The I/O-bound process is said to *steal* cycles from the CPU-bound process
- Multitasking
  - A mode of operation that provides for the concurrent performance, or interleaved execution, of two or more tasks
- Timesharing
  - A system in which two or more users share the CPU
  - Generally, the processing is quick so that the users do not notice that the CPU is being shared with other users
  - The CPU allocates a quantum of time (or time slice) to each user for processing before moving on to another user
- Multiprocessing
  - Use of two or more CPUs in a computer such that the CPUs have access to common storage (shared memory)

Concurrency at the Applications Level

- Interrupts

  - A suspension of processing caused by a deliberate instruction to the CPU
  - Usually done to allow the I/O operations to proceed
  - Each conventional machine level instruction executed in a processor instruction cycle
  - A device may generate a signal, called an interrupt, to set a hardware flag within the CPU
  - This flag is detected as a part of the instruction cycle by the CPU
  - When the interrupt is detected, the CPU saves the current value of the program counter register on stack and loads a new value in there which is the address of the *interrupt service routine*
  - After servicing the interrupt, CPU resumes the execution of the instruction it left off
  - Asynchronous or asynchronous interrupts
  - Asynchronous event
    * The time of occurrence is not determined by the entity that makes it happen
    * Interrupts generated by external hardware
    * Email received by you
    * The interrupts may not occur at the same point in the program
  - Synchronous events
    * Occurs depending on the data presented
    * Can be controlled
  - Interrupts generated by peripheral devices
  - Interrupts based on time to implement time sharing

- Signals
  - Software notification of an event
  - May be a response of the OS to a hardware event (interrupt)
  - The sequence of events associated with `^c`
    * User presses `^c`
    * Interrupt generated for the device driver handling the keyboard
    * Driver sends a signal to the appropriate process
    * Process commits suicide
  - Driver recognizes the character as an interrupt and notifies the process associated with the terminal by sending a signal
  - OS may also send a signal to a process to notify it of a completed I/O operation or an error
  - A signal is generated as a response to the occurrence of an event
  - A process *catches* a signal by executing a signal handler
  - Process and signal handler typically execute concurrently
    * Concurrency restricts what can be done inside signal handler
    * If signal handler modifies external variables that the program can modify elsewhere, proper execution my require those variables to be protected

- Input and output
  - Coordinate resources with different characteristic access times
  - Avoid blocking processes by using asynchronous I/O
  - Results in additional performance and extra programming overhead
  - What if a process waits for input from two different sources
    * A blocked wait for input from one source may miss input from the other source

- Threads and the sharing of resources
  - Multiple threads of execution provide concurrency within a process
  - A thread is a stream of instructions that define the flow of control for the process
  - Use of multiple threads with shared resources make the programming difficult
  - With multiprocessing systems, we can achieve multithreaded operation of a process
- The network as the computer
  - Distribution of computation over the net
  - Client-server model
    - Server process manage resources
    - Client processes use the resources by sending request to the server
  - Object-based model
    - Each resource is viewed as an object with a message handling interface
    - All shared resources are accesses in a uniform way
    - Object frameworks define interactions between code modules

System calls

- Interface between user program and operating system
- Provide a direct entry point into the kernel (privileged part of OS)
- Set of extended instructions provided by the operating system
- Applied to various software objects like processes and files
- Invoked by user programs to communicate with OS and request services
- Library functions
  - General purpose functions required in most programs
  - May invoke a system call to achieve the task
  - `fopen()` library function invokes `open()` system call
- Traditionally, system calls are described in section 2 of Unix manuals and library functions are described in section 3 of the manual
  - On delmar, you can get to a command in section n of the manual by invoking

```
man n command
```

- Whenever you use a system call or a library function, properly read the man page for the same, paying particular attention to the header files to be included
- Whenever a system call or library function encounters an error, it sends a signal back to the calling process who may decide to abort itself or continue
  - The system calls also sets an external variable `errno` to indicate the error code
  - This variable is not reset by the subsequent system calls which may execute successfully
  - You can include the file `errno.h` to access the symbolic error names associated with the error code
  - You can use the C library function `perror()` to display a message string to the standard error

  - The C library function `char * strerror( int );` returns a pointer to an error message string, given an `errno`

- Guidelines for good function development (based on system calls and C library functions)
  - Make use of return values to communicate information and to make error trapping easy for the calling program
  - Do not exit from functions; instead, return an error value to allow the calling program flexibility in handling the error
  - Make functions general but usable
  - Do not make unnecessary assumptions about sizes of buffers
  - When it is necessary to use limits, use standard, system-defined limits rather than arbitrary constants
  - Do not re-invent the wheel; use standard library functions when possible
  - Do not modify input parameter values unless it makes sense to do so
  - Do not use static variables or dynamic memory allocation if automatic allocation will do just as well
  - Analyze all the calls to the `malloc` family to make sure that the program frees all the memory that was allocated
  - Consider whether a function will ever be called recursively, or from a signal handler, or from a thread; reentrant functions are not self-modifying, so there can be simultaneous invocations active without interference; in contrast, functions with local static or external variables are nonreentrant and may not behave in the desired way when called recursively (the `errno` can cause a big problem here)
  - Analyze the consequence of interruptions by signals
  - Carefully consider how the program will terminate
- A Unix command line consists of tokens, with the first token (`argv[0]`) being the name of the command
- You can make an argument array from a string of tokens by using the function `makeargv`

`exec` system calls

- The only way to execute programs under Unix
- Reinitialize a process from a designated program
  - Program changes while the process remains
- Called by

```
int execl ( path, arg0, arg1, ..., argn, null )
```

  - All the arguments are of type `char *`, including `null`
  - The argument `path` must name an executable program file
  - The command

```
ls -l /bin
```

    is run by the system call

```
execl ( "/bin/ls", "ls", "-l", "/bin", NULL );
```

- Process's instruction segment is overwritten by the instructions from the program
- Process's user-data segment is overwritten by the data from the program
- Execution of the process begins at `main()`
- No return from a successful `execl` because the return location is gone

- Unsuccessful `execl` returns -1
  - Possible if the `path` does not exist, or is not executable
- The arguments can be collected by `argc` and `argv`
- Process continues to live and its system-data segment is largely undisturbed
  - All the process attributes are unchanged, including PID, PPID, process GID, real UID and GID, current and root directories, priority, accumulated execution times, and open file descriptors
  - Instructions designed to catch the signals need to be reexecuted as they are reset
  - If the SUID or SGID bit of the new program file is on, the effective UID or GID of the process is changed to reflect the same; former effective IDs cannot be retrieved
  - If the process was profiling, profiling is turned off
- Example

```
exectest()
{
    printf ( "The quick brown fox jumped over " );
    execl ( "/bin/echo", "echo", "the", "lazy", "dog.", NULL );
    printf ( "error in execl" );
}
```

  - The program may cause problem because of buffered I/O
  - Can be fixed by using `fflush(stdout);`
- If some file descriptor should not stay open across an `execl`, it must be explicitly closed
  - Wrong way to close file descriptors (assuming 20 file descriptors)

```
fdtest()
{
    for ( i = 0; i < 20; close ( i++ ) );
    execl ( path, arg0, arg1, arg2, NULL );
    printf ( "error in execl" );
}
```

    - `stderr` is also closed
  - Preferable way

```
fdtest()
{
    for ( i = 0; i < 20; fcntl ( i++, F_SETFD, 1 ) ); /* ignore errors */
    execl ( path, arg0, arg1, arg2, NULL );
    printf ( "error in execl" );
}
```

    - File descriptors are closed only on successful `execl`
- Other versions of `exec` – check the man pages

`fork` system call

- The only way to create new processes in Unix

  - Only exception provided by processes with PID 0, 1, and 2 which are created at bootstrapping time and are called *spontaneous processes*

- Create a new process that is a clone of the existing one
  - New process is called the *child process*
- Both parent and child continue execution at the instruction that follows the call to `fork`
- Copy the three segments (instructions, user-data, and system-data) without initialization from a program
- System call invoked by

```
int fork()          /* create new process */
/* return process-id and 0 on success, or -1 on error */
```

- Upon return, both parent and child receive the return
  - Child receives a 0 return value
    * 0 is not the PID of child because this is the PID of `swapper`
  - Parent receives the PID of the child
  - Usually, the child does an `exec` and the parent either waits for the child to terminate or goes off to do something else
  - Error occurs if there are no more resources
    * Insufficient swap space
    * Too many processes already in execution
- Child inherits most of the attributes from parent
  - Child's PID and PPID are different
  - Child gets copies of parent's open file descriptors
    * Each is opened to the same file and the file pointer has the same value
    * If the child changes the file pointer with `lseek`, parent's next read or write will be at the new location
    * File descriptor itself is distinct
      · If the child closes the file descriptor, the parent's copy is undisturbed
  - Child's accumulated execution times are reset to zero
  - Child and parent do not share portions of memory
- Example

```
forktest()
{
    int pid;
    printf ( "Start of test\n" );
    pid = fork();
    printf ( "Returned pid is: %d\n", pid );
}
```

`exit` system call

- Invoked by

```
void exit ( status )      /* terminate process */
int status;               /* exit status       */
```

- Terminates the process that issued it, with a status code equal to the rightmost byte of `status`
- All open file descriptors are closed
- All standard I/O streams are closed, and their buffers are flushed
- If child processes are still alive when `exit` is called, they are not disturbed
  - The PPID for such processes is changed to 1 (PID of `init`)
- The only system call that never returns
- By convention, a status code of zero means that the process terminated normally
- The exiting process's parent receives the status code through a `wait` system call

`wait` system call

- Invoked by

```
int wait ( statusp )     /* wait for child   */
int *statusp;            /* exit status      */
/* returns pid or -1 on error     */
```

- If there are many child processes, `wait` sleeps until one of them returns
- Caller cannot specify which child is to be waited for
  - Can be achieved by using the `waitpid` system call
- Process cannot receive a return from `wait` upon termination of a grandchild, those exit values are lost
- Process may terminate at a time when it is not being waited for
  - Kernel does not allow such processes to die
  - The unwaited for processes become *zombies*

Argument arrays

- Command line made up of tokens or arguments separated by whitespace
  - Whitespace is blank or tab or \ at the end of line
  - Each token a string of characters
  - No whitespace in a token unless protected by quotation marks
- Shell parses command line into tokens and passes the result to program in the form of an argument array
- Argument array is an array of pointers to strings
- End of array marked by an entry containing a `NULL` pointer
- Examples
  - Number of tokens in command

```
ls -l mydir
```

  - They are captured in `argv` by the program

- Creating an argument array with `makeargv`
  - Create the array from a string of tokens
  - Should take an input string parameter and return a pointer to an `argv` array
  - Returns the number of tokens in the input string
    - Indicate an error by -1
  - Prototype for the function

```
int makeargv ( char * s, char *** argvp );
```

  - The code should be used as

```
int     i;
char ** myargv;
char    mytest[] = "This is a test";
int     numtokens;

if ( ( numtokens = makeargv ( mytest, &myargv ) ) == -1 )
    fprintf ( stderr, "Failed to construct an argument array\n" );
else
    for ( i = 0; i < numtokens; i++ )
        printf ( "%d:%s\n", i, myargv[i] );
```

  - An even better prototype is

```
int makeargv ( const char * s, const char * delimiters, char *** argvp );
```

  - This code will be used as

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

int makeargv ( const char * s, const char * delimiters, char *** argvp );

int main ( int argc, char ** argv )
{
    int     i;
    char    delim[] = " \t";
    char ** myargv;
    int     numtokens;

    if ( argc != 2 )
    {
        fprintf ( stderr, "Usage: %s string\n", argv[0] );
        return ( 1 );
    }

    if ( ( numtokens = makeargv ( argv[1], delim, &myargv ) ) == -1 )
    {
        fprintf ( stderr, "Failed to construct an argument array for %s\n", argv[1] );
        return ( 1 );
    }

    printf ( "The argument array contains:\n" );
    for ( i = 0; i < numtokens; i++ )
        printf ( "%d:%s\n", i, myargv[i] );
```

```
    return ( 0 );
}
```

- Implementation of `makeargv`
  - Prototype given by

    ```
    int makeargv ( const char * s, const char * delimiters, char *** argvp );
    ```

  - No a priori assumption on the size of `s` or `delimiters`
    * A good idea to avoid imposing any arbitrary limit on buffer size
    * In case you must, use system-defined constant `MAX_CANON` for a buffer size for command line arguments
  - Release all dynamically allocated memory
  - No direct application of `strtok` on input string `s` to preserve the input string
  - Implementation strategy
    1. Use `malloc` to allocate a buffer `t` to parse input string; at least the same size to hold `s`
    2. Copy `s` to `t`
    3. Make a pass through `t` to count tokens using `strtok`
    4. Use the count to allocate an `argv` array
    5. Copy `s` into `t` again
    6. Use `strtok` to get pointers to individual tokens, modifying `t` and parsing `t` in place
  - A word on `strtok`
    * Prototype

      ```
      char * strtok ( char * restrict s1, const char * restrict s2 );
      ```

    * First call to `strtok` is different
    * On the first call, pass the address of the string to parse as the first argument; on subsequent calls, pass a `NULL` in its place
    * The second argument is a string of allowable token delimiters
    * Each successive call to `strtok` returns the start of next token and inserts a `'\0'` at the end of the token being returned
    * `strtok` returns `NULL` when there are no more tokens to be returned
    * `strtok` tokenizes the string in place; does not allocate new space for tokens
    * `restrict` qualifier on the two fprmal parameters requires that any object referenced by `s1` in this function cannot be accessed by `s2`
      · The tail end of `s1` cannot be used to contain the delimiters