Quiz:
What are the correct descriptions about system calls?
- System calls allow a program to request the service from the operating system.
- System calls are executed in user mode only.
- All APIs will involve in system calls finally.
- Typically, the library will provide wrappers for system calls.

Outline
- Programs, Processes and Threads
- Process creation and its components
- States of a process and transitions
- PCB: Process Control Block
- Process (program image) in memory
- Argument Arrays
- Making Functions Safe
- Storage and Linkage Classes

Objectives
- Introduce process concept -- program in execution, which forms the basis of all computation
- Understand: Program vs. Process vs. Threads
- Learn Various Aspects of Processes: Creation, State Transitions and Termination
- Learn the presentation of processes in OS: PCB (process control block)
- Understand Memory Layout of Program Image
- Understand Argument Array and Function Safety
- Understand Storage and Linkage Classes

Reviews and Updates
- Reviews
  - Course overview: syllabus and grading policies etc.
  - Course webpage: http://www.cs.utsa.edu/~tongpingliu/teaching/cs3733/cs3733.html
  - Email to contact me: “CS3733” in the subject line
- Updates
  - Recitation CS3733.004 (Wednesday)

Programs vs. Processes
- Program: a set instructions/functions
  - To accomplish a defined task
  - Passive entity, stored as files on disk
- Process: a program in execution
  - Dynamic concept: running of a program
  - Unit of work in a system
- Multiple processes may be from a single program
  - Run/executing the program multiple times

How do we run/execute a program?
“top” utility to show processes

```
User      %CPU   %MEM    SWAP    PID  PPID   PRI   NICE  ADDR   COMMAND
root      0.00   1.00    0.00   18332 0     40    0   0.00   0.00   top
```

Process: Address Space

- A process includes:
  - Program code: text segment
  - Global variables: data section
  - Temporary data (local variables, function parameters, and return address etc.): Stack
  - Dynamically allocated data (malloc): Heap
  - Program counter (PC) and registers

- Process execution must progress in sequential fashion (a single thread)

An Example Program

```
int foo(int x)
{
    return x;
} // foo is popped off the call stack here

int main()
{
    int *ptr = malloc(sizeof(int));
    foo(5); // foo is pushed on the call stack here
    return 0;
}
```

Process States

- As a process executes, it changes its state
  - New: being created and starting up
  - Running: instructions being executed
  - Waiting: waiting for some event (such as I/O) to occur
  - Ready: waiting to be assigned to a processor
  - Terminated/Halted: finished execution

- CPU switch from a process to another one

State Transitions of A Process

- Possible state transitions for a process:

  - What transitions are impossible/invalid?!
PCB: Process Control Block

- OS manages processes via their PCBs
- PCB of a process consists of:
  - Process state (one of above states)
  - Program counter (PC)
  - CPU registers (necessary registers to re-start the process)
    - Stack pointer (SP): top of current stack
    - Program status word (PSW): condition code bits and control bits
  - CPU scheduling information: e.g., priority
  - Memory management: page table location
  - Accounting information: CPU time used
  - I/O status information: I/O requests; open files; I/O devices

Process Creation in UNIX

- Process has a process identifier (pid)
- A process (parent) creates another process (child) using the system call fork
- The new child process has a separate copy of the parent process’s address space (code, data etc.).
- Both parent and child processes continue execution at the instruction right after the fork system call
  - Return value of 0 \( \rightarrow \) new (child) process continues
  - Otherwise, return non-zero pid of child process \( \rightarrow \) parent process continues

An Example: \texttt{fork( )} in UNIX

- pid = 25
- pid = 26
- \texttt{int cpid = fork( );}
- \texttt{if (cpid = = 0) {<child code>}}
- \texttt{exit(0);}}
- \texttt{<parent code> wait(cpid);}
- \texttt{return (cpid);}
- \texttt{wait(cpid);}

Process vs. Threads

- A process stars with a single flow of control
- The flow executes a sequence of instructions: thread of execution
- Thread of execution: logically related sequence of instruction address from PC during execution
- Thread: represents a thread of execution of a process \( \rightarrow \) basic unit of CPU utilization
  - Thread ID
  - PC
  - Register set
  - Stack

Example 2.2 from USP Book

- Suppose
  - process 1 executed statements 245, 246 and 247 in a loop
  - process 2 executes the statements 10, 11, 12, 13, …
  - CPU starts executing process 1 for 5 instructions;
  - process 1 loses CPU;
  - CPU then executes 4 instructions of process 2 before losing the CPU; the executed sequence of instructions:
    - 245, 246, 247, 245, 246, 10, 11, 12, 13, 247, 245, 246, …; subscript indicates which process
- Two threads of execution (each from a process):
  - 245, 246, 247, 245, 246, 247, … And
  - 10, 11, 12, 13,

Program Image in Memory (Fig. 2.1 USP)

- \texttt{int cpid = fork( );}
- \texttt{if (cpid = = 0) {<child code>}}
- \texttt{exit(0);}}
- \texttt{<parent code> wait(cpid);}
- \texttt{return (cpid);}
- \texttt{wait(cpid);}
Argument Arrays

Argument array

- an array of pointers terminated by a NULL pointer
- Each element is of type char * and represents a string.

\[
\text{argv}[]
\]

[0] — "s" | "0" |
[1] — "-" | "0" |
[2] — "1" | "0" |
[3] — "2" | "0" |
[4] — NULL

Figure 2.2 (page 32): The argv array for the call \texttt{mine \textasciitilde c 10 2.0}.

Create Your Own Argument Arrays

- Create an argument array from a string
- \texttt{argv[]} is an array of pointers to strings;
- \texttt{char **makeargv(char *s)}

To return number of tokens

- \texttt{int makeargv(char *s, char ***argvp)}

String of delimiters

- \texttt{int makeargv(const char *s, const char *delimiters, char ***argvp)}
- The \texttt{const} for the first two parameters indicates that the strings should not be modified by the function.

An example to use \texttt{makeargv()}:

Program 2.1: args2.c, page 34

\begin{verbatim}
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int i;
    char **argvp;
    int ntokens;
    if (argc < 2) {
        printf("Usage: sample
argv[0] [options] [argv[1] ... argv[n-1]]
");
        return 1;
    }
    argvp = makeargv(argv, &ntokens);
    if (ntokens > 0) {
        printf("The number of arguments passed to makeargv was %d
", ntokens);
    } else {
        printf("The number of arguments passed to makeargv was 0
");
    }
    return 0;
}
\end{verbatim}

How to Write \texttt{makeargv()}:

- Use the function \texttt{strtok()}:
  - \texttt{#include <string.h>}
  - \texttt{char *strtok(char *restrict s1, const char *restrict delimit)};
  - \texttt{s1} is the string to parse; Note that, \texttt{s1} will be modified;
  - \texttt{delimit} is a string of delimiters
  - \texttt{To NOT modify the string passed to makeargv()}:
    - Get a copy of the string;
    - Make a pass with \texttt{strtok} to count the tokens
    - Use the count to allocate the \texttt{argv} array

- Second pass with \texttt{strtok} to set pointers in \texttt{argv}

The Function: \texttt{makeargv()}

- copy of the string
- Count tokens
- Allocate argvp
- Set pointers in argvp

Result Array of The Example

Figure 2.4 (page 36): The use of \texttt{strtok} to allocate strings in place for \texttt{makeargv}.
Safe Issues of `strtok()`

- `strtok()` is not safe to use with threads
  - it remembers the previous state
- Programs using `strtok` can fail even w/o threads
  - Suppose to write a program that calculates the average number of words per line in a text file
    - `double wordaverage(char *s);`
    - Parse input string into lines using `strtok`;
    - then call a function: `int wordcount(char *s);`
    - If `wordcount()` also uses `strtok()`, program fails!

Making Functions Safe

- A safe version of `strtok`: `strtok_r`

  ```c
  #include <string.h>
  char *strtok_r(char *restrict s1, const char *restrict s2, char **restrict lasts);
  ```

  - A char pointer to hold the current position

- Use `strtok_r()` in either `wordaverage()` or `wordcount()` will solve above problem!

Storage Classes (USP Appendix A.5)

- **Static vs. automatic**
  - Static storage class: variables that, once allocated, persist throughout the execution of a program
  - Automatic storage class: variables which come into existence when block in which they are declared is entered; discarded when the defining block is exited

- Variables
  - Variables defined outside any functions have **static** storage class.
  - Declared inside a function have **automatic** storage class (unless they are declared to be static), which are usually allocated on the program stack

Linkage Classes

- **static** has two meanings in C
  - One related to storage class
  - The other to linkage class

- Linkage classes: determines whether variables can be accessed in files other than the one in which they are declared
  - **Internal** linkage class: can only be accessed in the file in which they are declared
  - **External** linkage class: can be accessed in other files.
Linkage Classes (cont.)

- Variables
  - Declared outside any function and function name identifiers have external linkage by default; however, they can be given internal linkage with the keyword `static`.
  - Declared inside a function are only known inside that function and are said to have no linkage.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>When Declared</td>
<td>Static Modifiers</td>
</tr>
<tr>
<td>Inside a function</td>
<td>storage class</td>
</tr>
<tr>
<td>Outside any function</td>
<td>storage class</td>
</tr>
</tbody>
</table>

Example Program: bubblesort.c

```c
#include <stdio.h>

#define N 10

int main(void)

int i, j, k, n; /* n (number of elements) */

int array[N]; /* array of n elements */

int count = 0;

for (i = 0; i < n; i++)

for (j = 0; j < n - 1; j++)

if (array[i] > array[j])

swap(array[i], array[j]);

count++;

return count;

int swap(void) {

int temp = 0;

temp = array[i];

array[i] = array[j];

array[j] = temp;

return temp;

}
```

Summary

- Programs, Processes and Threads
- Process creation and its components
- States of a process and transitions
- PCB: Process Control Block
- Process (program image) in memory
- Argument Arrays
- Making Functions Safe
- Storage and Linkage Classes

Programs vs. Processes

- Program: a set instructions/functions
  - To accomplish a defined task
  - Passive entity, stored as files on disk
- Process: a program in execution
  - Dynamic concept: running of a program
  - Unit of work in a system
- Multiple processes may be from a single program
  - Run/execute the program multiple times

State Transitions of A Process

- Possible state transitions for a process:

```
new
admitted
interrupt
exit
terminated

ready
running
waiting
I/O or event completion
scheduler dispatch
I/O or event wait
```

Process: a program in execution
PCB: Process Control Block

- OS manages processes via their PCBs
- PCB of a process consists of:
  - Process state (one of above states)
  - Program counter (PC)
  - CPU registers (necessary registers to re-start the process)
    - Stack pointer (SP): top of current stack
    - Program status word (PSW): condition code bits and control bits
  - CPU scheduling information: e.g., priority
  - Memory management: page table location
  - Accounting information: CPU time used

I/O status information: I/O requests; open files; I/O devices.

Process: Address Space

- A process includes:
  - Program code: text segment
  - Global variables: data section
  - Temporary data (local variables, function parameters, and return address etc.): Stack
  - Dynamically allocated data (malloc): Heap
  - Program counter (PC) and registers

- Process execution must progress in sequential fashion (a single thread)

An Example Program

```c
1: int main()
2: {
3:    int *ptr = malloc(sizeof(int));
4:    fprintf(stderr, "ptr is at %p, ptr is %p\n", &ptr, ptr);
5:    return 0;
}
```

Storage Classes (USP Appendix A.5)

- **Static vs. automatic**
  - Static storage class: variables that, once allocated, persist throughout the execution of a program
  - Automatic storage class: variables which come into existence when block in which they are declared is entered; discarded when the defining block is exited

Variables

- Variables defined outside any functions have **static** storage class.
- Declared inside a function have **automatic** storage class (unless they are declared to be static), which are usually allocated on the program stack

Linkage Classes

- **static** has two meanings in C
  - One related to storage class
  - The other to linkage class

Linkage classes: determines whether variables can be accessed in files other than the one in which they are declared

- Internal linkage class: can only be accessed in the file in which they are declared
- External linkage class: can be accessed in other files.

Linkage Classes (cont.)

- **Variables**
  - Declared outside any function and function name identifiers have external linkage by default; however, they can be given internal linkage with the key word **static**
  - Declared inside a function are only known inside that function and are said to have **no linkage**
Example Program: bubblesort.c

```
#include <stdio.h>

static int count = 0;

void main()
{
  printf("L0\n");
  int n = 10;
  for (int i = 0; i < n; i++)
  {
    printf("L1\n");
    for (int j = 0; j < n; j++)
    {
      printf("L2\n");
      count = count + 1;
      if (j == n - 1)
        break;
    }
    printf("Bye\n");
  }
}
```

Outline

- Examples on process creation and termination
- Types of processes: CPU-Bound and IO-Bound

Create & Terminate Processes

What does this print out?

```
void main()
{
  printf("L0\n");
  fork();
  printf("L1\n");
  fork();
  printf("L2\n");
  fork();
  printf("Bye\n");
}
```

Does it always print in order?

```
void main()
{
  printf("L0\n");
  if (fork() != 0) {
    printf("L1\n");
    if (fork() != 0) {
      printf("L2\n");
      fork();
    }
  }
  printf("Bye\n");
}
```

Create & Terminate Processes

What happens here?

```
void main()
{
  if (fork() == 0) {
    /* Child */
    printf("Terminating Child, PID = %d\n",
           getpid());
    exit(0);
  } else {
    printf("Running Parent, PID = %d\n",
           getpid());
    while (1)
      ;
  }
}
```

04_fork.c

05_fork.c

05_zombie.c
Linux

```
void main()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
                getpid());
        while (1)
            /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
                getpid());
        exit(0);
    }
}
```

Really hard to detect since the main process already exited!

How do we control processes?

Need a way to kill the zombies!

This is called reaping!

So, how do we “reap” a child process programmatically?

Create & Terminate Processes

What about this one?

Zombie?
Create & Terminate Processes

So, how do we “reap” a child process programmatically?

wait()
waitpid()

Status Values for wait

- Status value == 0 if and only if the process terminated normally and returned 0
- Other cases: the status should be examined using macros defined in sys/wait.h

```
#include <sys/wait.h>

WIFEXITED(int stat_val)
WEXITSTATUS(int stat_val)
WIFSIGNALED(int stat_val)
WTERMSIG(int stat_val)
WIFSTOPPED(int stat_val)
WSTOPSIG(int stat_val)
```

Used in pairs

Create & Terminate Processes

```
int wait(int* child_status)
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
    {
        if ((pid[i] = fork()) == 0)
            exit(100+i);
        /* Child */
        for (i = 0; i < N; i++)
        {
            pid_t wpid = wait(&child_status);
            if (WIFEXITED(child_status))
                printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
            else
                printf("Child %d terminated abnormally\n", wpid);
        }
    }
}
```

Create & Terminate Processes

```
int wait(id, &status, options)
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
    {
        if ((pid[i] = fork()) == 0)
            exit(100+i);
        /* Child */
    }
    for (i = 0; i < N; i++)
    {
        pid_t wpid = wait(id, &child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```
Create & Terminate Processes

\[
\text{int waitpid(-1, \&status, 0)}
\]

is the same as...

\[
\text{int wait(\&status)}
\]

Activity

List all the possible output sequences for this program.

```c
int wait(pid_t pid, int* statptr = NULL, int options = 0); int wait(int* statptr = NULL, int options = 0); int wait(const siginfo_t *info, int options = 0); int waitpid(pid_t pid, int *wstatus = NULL, int options = 0); int wait4(pid_t pid, int *wstatus, int options, void **child_ptr); int wait3(int options, struct siginfo *infop, struct pt_regs *statp); int wait4(int options, struct siginfo *infop, struct pt_regs *statp, void **child_ptr);
```

Activity

List all the possible output sequences for this program.

```c
data main()
{
    if (fork() == 0) {
        printf("a");
    } else {
        printf("b");
        waitpid(-1, NULL, 0);
        printf("c");
        exit(0);
    }
}
```

Solution:
We can't make any assumption about the execution order of the parent and child. Thus, any topological sort of b -> a and a -> c is possible:

- acbc
- abcc
- bacc

Process Termination

- Voluntarily
  - process finishes and asks OS to delete it (exit).
- Involuntarily
  - parent terminate execution of children processes (e.g. TerminateProcess() in Win32).
  - A process may also be terminated due to errors, segv

- After process terminate
  - Output data from child to parent (wait or waitpid).
  - Process' resources are de-allocated by OS.

- Parent process is terminated (e.g., due to errors) Zombies

Performance metrics of a process

- Performance metrics of evaluating a process:
  - Throughput: the amount of work to be finished in a time unit
  - Utilization: the fraction of a system is busy with useful work
  - Turnaround time: the time from start to completion, including
  - Waiting time to be loaded into memory,
  - Waiting time in ready queue,
  - Execution time
  - Blocked time waiting for an event, for example waiting for I/O
  - Response time: the time from arrival time of a request to the time its response is produced
  - Waiting time: the delay time in Ready queue, directly impacted by scheduling discipline
  - Fairness: each process/thread gets fair share of resources (cpu time, etc)
  - Deadline in real-time scheduling
  - etc.
- Maximize/minimize metrics,
  - Bounded vs average result; Deterministic vs stochastic

Borrow this slide from Dr. Xu at Wayne State

CPU-I/O bursts

- process execution consists of a cycle of CPU execution and I/O wait
  - different processes may have different distributions of bursts
  - CPU-bound process: performs lots of computations in long bursts, very little I/O
  - I/O-bound process: performs lots of I/O followed by short bursts of computation
  - ideally, the system admits a mix of CPU-bound and I/O-bound processes to maximize CPU and I/O device usage

- CPU burst
- I/O burst
- I/O burst
- I/O burst
Activities in Processes

- Bursts of CPU usage alternate with periods of I/O wait
- **CPU-bound**: high CPU utilization, interrupts are processed slowly
- **I/O-bound**: more time is spent on requesting data than processing it

Multiprogramming

- Multiprogramming is a form of parallel processing in which several programs are run at the same time on a uniprocessor.
  - Objective?

  Maximize CPU utilization. When a process waits for I/O, all waiting time is wasted and no useful work is accomplished.

Burst distribution

- CPU bursts tend to have an exponential or hyper-exponential distribution
  - there are lots of little bursts, very few long bursts
  - a typical distribution might be shaped as here:

Assignment

- See http://www.cs.utsa.edu/~tongpingliu/teaching/cs3733/assign0.html

Quiz

```c
// Given program segment
int c2 = 0;  // fork number 1
if (c1 == 0) {
    c2 = fork();  // fork number 2
    if (c2 > 0) {
        fork();  // fork number 3
        if (c2 > 0) {
            fork();  // fork number 4
        }
    }
}
```