CS3733: Operating Systems

Topics: Programs and Processes
(SGG 3.1-3.2; USP 2)

Instructor: Dr. Tongping Liu
Fall 2016

Reviews and Updates

- Reviews
  - Course overview: syllabus and grading policies etc.
  - Course webpage: [http://www.cs.utsa.edu/~tongpingliu/teaching/cs3733/cs3733.html](http://www.cs.utsa.edu/~tongpingliu/teaching/cs3733/cs3733.html)
  - Email to contact me: "CS3733" in the subject line

- Updates
  - Recitation CS3733.004 (Wednesday)

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
- Storage and Linkage Classes
- Process Creation and Termination

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

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

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

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PRI</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
</tr>
</thead>
<tbody>
<tr>
<td>2766</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>53983</td>
<td>51736</td>
<td>11328</td>
<td>0.3</td>
<td>1.2</td>
<td>156:12.12</td>
</tr>
<tr>
<td>4159</td>
<td>daemon</td>
<td>20</td>
<td>413566</td>
<td>53983</td>
<td>51736</td>
<td>11328</td>
<td>0.3</td>
<td>1.2</td>
<td>156:12.12</td>
</tr>
<tr>
<td>11090</td>
<td>tom@lin</td>
<td>20</td>
<td>5372</td>
<td>5372</td>
<td>5</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:00.28</td>
</tr>
</tbody>
</table>

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

int main()
{
    int * ptr = (int *) malloc(sizeof(int));
    fprintf(stderr, "ptr is at %p, ptr is %p\n", &ptr, ptr); // ptr is 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:
  - new
  - admitted
  - interrupt
  - wait
  - terminate

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
  - 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 — new (child) process continues
- Otherwise, return non-zero pid of child process — parent process continues

An Example: fork() In UNIX
- pid = 25
- pid = 26
- pid = 0

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, ...
  - Two threads of execution (each from a process):
    - 245, 246, 247, 245, 246, 247, And
    - 10, 11, 12, 13,
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 the 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**

```c
. The function onepass has internal linkage;
. The other functions have external linkage;
. Functions do not have a storage class;
. The count variable has internal linkage and static storage;
. All other variables have no linkage and automatic storage.
```

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

Review of Last Class
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

**How do we run/execute a program?**

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

State Transitions of A Process

- Possible state transitions for a process:
  - Admitted
  - Ready
  - Running
  - I/O or event completion
  - Waiting
  - Exit
  - Terminated

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 = (int *)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 determine 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.

Example Program: **bubblesort.c**

```c
void main()
{
    int count = 0;
    static int comp1[30]; int a, b; /* return true if interchange are made */
    int interchange = 0;
    for (a = 0; a < n; a++) { /*
        for (b = 0; b < n; b++)
            if (a < b && a < b)
            
        interchange += 1;
    }
    return interchange;
}
```

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
- Storage and Linkage Classes
- Process Creation and Termination

Create & Terminate Processes

**What does this print out?**

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

Create & Terminate Processes

**What does this print out?**

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

**Does it always print in order?**

04_fork.c
Create & Terminate Processes

What does this print out?

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

What happens here?

```c
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) ; /* Infinite loop */
    }
}
```

What about this one?

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

Really hard to detect since the main process already exited!

Need a way to kill the zombies!
How do we control processes?

Need a way to kill the zombies!

This is called reaping!

Create & Terminate Processes

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

wait()

waitpid()

Create & Terminate Processes

int wait(int* child_status)

fork a child processes

wait for each to terminate

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

int wait(int* child_status)

fork a child processes

wait for each to terminate
Create & Terminate Processes

int wait(int* child_status)

```c
void main()
{
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++)
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 waitpid(pid, &status, options)

```c
void main()
{
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 = N-1; i >= 0; i--)
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 waitpid(-1, &status, 0)

is the same as...

int wait(&status)

Activity

List all the possible output sequences for this program.

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

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

List all the possible output sequences for this program.

- acbc
- abcc
- bacc
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,
  - 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

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

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 spending 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.

Maximize CPU utilization. When a process wait 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:

What does this distribution pattern imply about the importance of CPU scheduling?

Quiz

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

fork2 fork3