**CS3733: Operating Systems**

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

**Instructor:** Dr. Tongping Liu

---

**Outline**

- Programs and Processes
- States of a process and transitions
- PCB: Process Control Block
- Process (program image) in memory
- 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 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

Components of A Process

1. 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

2. Process execution must proceed in the **sequential** fashion (for each thread of execution)

---

**An Example Program**

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

```c
int main()
{
    int *ptr = malloc(sizeof(int));
    //foo is pushed on the call stack here
    return 0;
}
```
Programs and Processes
States of a process and transitions
PCB: Process Control Block
Process (program image) in memory
Process Creation and Termination

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, exit, terminated
  - Ready, running, I/O or event completion, scheduler dispatch, waiting
  - I/O or event wait

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
  - Accounting information: CPU time used
  - I/O status information: I/O requests; open files; I/O devices

---

**Running Context of A Process**

- Registers: in addition to general registers
  - Program Counter (PC): contains the memory address of the next instruction to be executed.
  - Stack Pointer (SP): points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited.
- Higher level resources: open files etc.
- Synchronization and communication resource: semaphores and sockets

---

**Thread of Execution**

- A process starts 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

---

**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, ...
Outline

- Programs and Processes
- States of a process and transitions
- PCB: Process Control Block
- Process in Memory
- Process Creation and Termination

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.

<table>
<thead>
<tr>
<th>Where Declared</th>
<th>static Modeled</th>
<th>Storage Class</th>
<th>Linkage Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>inside a function</td>
<td>storage class</td>
<td>static</td>
<td>none</td>
</tr>
<tr>
<td>inside a function</td>
<td>linkage class</td>
<td>static</td>
<td>internal</td>
</tr>
<tr>
<td>outside any function</td>
<td>linkage class</td>
<td>static</td>
<td>internal</td>
</tr>
<tr>
<td>outside any function</td>
<td>linkage class</td>
<td>static</td>
<td>internal</td>
</tr>
</tbody>
</table>

Example Program: bubblesort.c

```c
#include <stdio.h>

int main() {  
    int count = 0;  
    for (int i = 0; i < n; i++) {  
        if (count == i) {  
            printf("count = %d
", count);  
        }  
        count++;  
    }  
    return 0;  
}
```

Outline

- Programs and Processes
- States of a process and transitions
- PCB: Process Control Block
- Process in Memory
- Process Creation and Termination

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

In UNIX

```c
int cpid = fork();
if (cpid == 0) {
    <child code>
    exit(0);
} //parent code
wait(cpid);
```

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

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");
        }
    }
printf("Bye\n");
}
```

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");
}
```
### Create & Terminate Processes

#### 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 */
    }
}
```

---

### Linux

Linux

#### 05_zombie.c

- `/bin/Exceptions and Processes Part 1 Code`:
  - Running Parent, PID = 20291
  - Terminating Child, PID = 20202
- `/bin/Exceptions and Processes Part 1 Code`:
  - `ps` output shows:
    - `ps` process
    - `bash` process

---

#### What about this one?

```c
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);
    }
}
```

---

### 06_zombie.c

- `/bin/Exceptions and Processes Part 1 Code`:
  - Running Child, PID = 20203
- `/bin/Exceptions and Processes Part 1 Code`:
  - `ps` output shows:
    - `ps` process
    - `bash` process

---

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!

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)

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++) {
        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);
    }
}
```
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--)
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        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);
}
```

wait for PID to terminate

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\n");
    }
    else {
        printf("b\n");
        waitpid(-1, NULL, 0);
    }
    printf("c\n");
    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:

abbc
acbc
bacc
bc

Activity

List all the possible output sequences for this program.

```c
void main()
{
    if (fork() == 0) {
        printf("a\n");
    }
    else {
        printf("b\n");
        waitpid(-1, NULL, 0);
    }
    printf("c\n");
    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, e.g., `segv`.

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

**Parent process is terminated (e.g., due to errors)**
- What will happen to the children process?!