Programs and Processes

(SGG 3.1-3.2; USP 2)

Instructor: Tongping Liu

Outline
- Programs and Processes
- Process States and State Transitions
- Process Control Block
- Process in the Memory
- Process Creation and Termination

Objectives
- Introduce the process concept, which forms the basis of all computation
- Understand: Programs vs. Processes
- Learn Various Aspects of Processes: Creation, State Transitions, and Termination
- Learn the presentation of processes in OS: PCB (process control block)
- Understand The Memory Layout of Program
- Understand Storage and Linkage Classes

Programs vs. Processes
- A program consists of 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

```
User      PID         NI  VIRT  RES  SHR  %CPU %MEM    TIME+ COMMAND
root      2         0  135504  31792  3216  80  0.0  0.3  15:37.24 redhat
10110 daemon  25  0  41664  13560  5596  11  0.3  0.1  31:15.27 rc堕.bnm
10590 klogd   38  0  21844  3793  3658  85  0.0  0.6  18:40.87 klog
10  root      148232  0  5628  3752  3162  0  0.0  0.0  11:53.53 systemd
3  root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
5  root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
7  root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
8  root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
9  root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
10 root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
11 root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
12 root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
13 root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
14 root      148232  0  20222  5928  3162  0  0.0  0.0  11:53.53 systemd
```

Process Information
- PD: process ID
- USER: who starts this process
- PR: priority
- VIRT: virtual memory
- RES: resident physical memory
- SHR: shared size
- %CPU: percentage of CPU utilization (> 100% in multicore machine)
- %MEM: percentage of Memory utilization

Outline
- Programs and Processes
- Process States and State Transitions
- Process Control Block
- Process in the Memory
- Process Creation and Termination

Process States
- As a process executes, it changes its **state**
  - **New**: being created
  - **Running**: instructions being executed
  - **Waiting**: waiting for some events (such as I/O) to occur
  - **Ready**: waiting to be assigned to a processor
  - **Terminated/Halted**: finished execution
- CPU is switched from a process to another one
State Transitions of A Process

Possible state transitions for a process:

- New
- Admitted
- Running
- Ready
- Waiting
- Exit
- Terminated

What transitions are impossible/invalid?!

Outline

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

PCB: Process Control Block

- OS manages processes via their PCBs
- PCB of a process consists of:
  - Process state (one of above states)
  - Process number (ID)
  - Program counter (PC)
  - CPU registers:
    - Stack pointer (SP): top of current stack
    - General registers
  - CPU scheduling information: e.g., priority
  - Memory management information
  - Accounting information: CPU time used
  - I/O status information: I/O requests; open files; I/O devices

Example: PCB in Linux (task_struct)

- Process Management
  - Registers
  - Program Counter
  - Stack Pointers
  - Process State
  - Priority
  - Scheduling Parameters (slice)
  - Process ID
  - Parent process
  - Process group
  - Time when process started
  - CPU time used

- Memory Management
  - Pointer to text (code) segment
  - Pointer to data segment
  - Pointer to stack segment

- File Management
  - Root directory
  - Working directory
  - User ID
  - Group ID
  - List of open files
Running Context of A Process

- Process context is basically the process’s current state (what is in its registers).

- **Special 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.

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 to the sequence of instruction addresses 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, ...

PCB is Used in Context Switches
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

- Running context of a process is represented in the PCB.

- Context-switch time is overhead.

- Hardware support:
  - Multiple set of registers.

- Other performance issues/problems:
  - Cache content: locality is lost.
  - TLB content: may need to flush.

Outline

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

Components of A Process

- 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 proceed in the sequential fashion (for each thread of execution).

Address Space

- address space is the set of ranges of virtual addresses that an operating system makes available to a process.

- How big is the address space?
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));

    foo(5); // foo is pushed to the call stack
    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 that are allocated when a block in which they are declared is entered; discarded when the defining block is exited

- **Variables storage class**
  - 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/C++
  - One related to storage class
  - The other related 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 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 Modifiers</th>
<th>Storage Class</th>
<th>Linkage Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>inside a function</td>
<td>yes static none</td>
<td>internal</td>
<td>none</td>
</tr>
<tr>
<td>outside any function</td>
<td>no public protected</td>
<td>internal</td>
<td>no internal</td>
</tr>
<tr>
<td>inside a function</td>
<td>no public protected</td>
<td>internal</td>
<td>no internal</td>
</tr>
</tbody>
</table>

Example Program: `bubblesort.c`

```c
static int count = 0;
static int swapcount = 0;

int main(int argc, char *argv[]) {
    int n;
    int i;
    int j;
    int a[100];
    for (i = 0; i < n; i++)
        a[i] = i;
    count = 0;
    swapcount = 0;
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) {
                swap(a[i], a[j]);
                count++;
                swapcount++;
            }
    return 0;
}
```

Outline

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

Running Context of A Process

- Process context is basically the process’s current state (what is in its registers).
- **Special 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.
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 their execution right after the **fork** system call
  ➢ Return value of 0 → new (child) process continues
  ➢ Otherwise, return a pid of child process (>0) → parent process continues
  ➢ When return value is less than 0, indicates a failing fork

**An Example:** fork() in UNIX

```c
int cpid = fork();
if (cpid == 0) {
    // child code
    exit(0);
} else if (cpid > 0) {
    // parent process
    wait(cpid);
} else { // parent will wait for child to complete
    exit(0);
}
```

When fork() may fail (**man 2 fork**)?

- **EAGAIN**
  ➢ Cannot allocate memory to copy the parent’s page tables and allocate a task structure for the child.
  ➢ Encounter the limit of active process
  ➢ In a scheduling state (**SCHED_DEADLINE**)
- **ENOMEM**
  ➢ Failed to allocate the necessary kernel structures
- **ENOSYS**
  ➢ fork() is not supported, e.g. Hardware has no MMU

Load A Different Program

- Child process uses **exep** to load a different program

```c
void main () {
    int pid;
    pid = fork();
    if (pid < 0) { error_msg; }
    else if (pid == 0) { /* child process */
        exec ("/bin/ls", "ls");
        exit();
    } else { /* parent process */
        wait(NULL);
        exit(0);
    }
}
```
Create & Terminate Processes

What does this print out?

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

L0
L1
Bye
Bye
Bye
Bye

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");
}
```

L0
L1
L2
L2
Bye
Bye
Bye
Bye

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) ;
    }
}
```

Zombie!
A zombie process or defunct process is a process that has completed execution (via the `exit` system call) but still has an entry in the process table, with the “Terminated State”.

- Occurs for child process
- The entry is still needed to allow the parent process to read its exit state: once the exit status is read via `wait`, the zombie's entry is removed from the process table and it is said to be "reaped".

Problems:
- Cause resource leak

---

Create & Terminate Processes

What about this one?

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

---

Zombie? 06_zombie.c

A zombie process is a process that has completed execution, but whose parent has died.

Orphan process: 06_zombie.c

Really hard to detect since the main process already exited!

A orphan process is a process that is still executing, but whose parent has died.
Create & Terminate Processes

What about this one?

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

No problem!

Handle Zombie and Orphan Processes in Linux

- Zombies may cause resource leak, such as memory and process table entries (PIDs)
- Orphan processes will be reaped by init process (process PID 1), which waits on all of children

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?

Zombie?

About wait() (man 2 wait)

- **Wait()**: wait for state changes in a child of the calling process, and obtain information about the child whose state has changed
- Multiple conditions of a state change:
  - The child terminated; release resources of the child
  - The child was stopped by a signal
  - The child was resumed by a signal.
- If a wait is not performed, then the terminated child remains in a “zombie” state
  - Can a process wait for a random process?

Status Values for wait

- Status value == 0 if and only if the process terminated normally and returned 0
- Other cases: the status can 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)
  WIFSTOPSIG(int stat_val)
  ```
- Used in pairs with wait()
int waitpid(-1, &status, 0) is the same as...
int wait(&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 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)

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);
    }


Create & Terminate Processes

wait for PID to terminate

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
  - Process' resources are de-allocated by OS.
- Parent process is terminated (e.g., due to errors)
  - What will happen to the children process?!

Orphans