Process

Heechul Yun

Disclaimer: some slides are adopted from the book authors’ slides with permission
Recap

• OS services
  – Resource (CPU, memory) allocation, filesystem, communication, protection, security, I/O operations

• OS interface
  – System-call interface

• OS structure
  – Monolithic, microkernel
  – Loadable module
Roadmap

• Beginning of a series of important topics:
  – Process
  – Thread
  – Synchronization

• Today
  – Process concept
  – Context switching
Process

- Process
  - An OS abstraction represents a running application

- Three main components
  - Address space
    - The process’s view of memory
    - Includes program code, global variables, dynamic memory, stack
  - Processor state
    - Program counter (PC), stack pointer, and other CPU registers
  - OS resources
    - Various OS resources that the process uses
    - E.g.) open files, sockets, accounting information
Process Address Space

- **Text**
  - Program code

- **Data**
  - Global variables

- **Heap**
  - Dynamically allocated memory
    - i.e., Malloc()

- **Stack**
  - Temporary data
  - Grow at each function call
Process Address Space

• Each process has its own **private** address space
  – \(2^{32}\) (4GB) of **continuous memory** in a 32bit machine
  – Each has same address range (e.g., 0x0 ~ 0xffffffff)
  – How is this possible?
    • What if you have less than 4GB physical DRAM?
    • What if you have 100 processes to run?

• **Virtual memory**
  – An OS mechanism providing this **illusion**
  – We will study it in great detail later in the 2\(^{nd}\) half of the semester
Virtual Memory vs. Physical Memory

Virtual Memory

Process A
- Stack
- Heap
- Data
- Text

Process B
- Stack
- Heap
- Data
- Text

Process C
- Stack
- Heap
- Data
- Text

Physical Memory
– **running**: Instructions are being executed
– **waiting**: The process is waiting for some event to occur
– **ready**: The process is waiting to be assigned to a processor
Process Control Block (PCB)

- Information associated with each process
  - Process id
  - Process state
    - running, waiting, etc.
  - Saved CPU registers
    - Register values saved on the last preemption
  - CPU scheduling information
    - priorities, scheduling queue pointers
  - Memory-management information
    - memory allocated to the process
  - Accounting information
    - CPU used, clock time elapsed since start, time limits
  - OS resources
    - Open files, sockets, etc.
Process in Linux

Represented by the C structure `task_struct` (include/linux/sched.h)

```c
struct task_struct
{
   pid_t pid;    /* process identifier */
   long state;    /* state of the process */
   u64 vruntime;   /* CFS scheduling information */
   struct files_struct *files; /* list of open files */
   struct mm_struct *mm;  /* address space of this process */
   cputime_t utime, stime; /* accounting information */
   struct thread_struct thread; /* CPU states */

   /* ... */

   (very big structure: 5872 bytes in my desktop *)
}
```

(*) # cat /sys/kernel/slab/task_struct/object_size
Process Scheduling

• Decides which process to run **next**
  – Among **ready** processes

• We cover in much more detail later in the class
  – but let’s get some **basics**

• OS maintains multiple **scheduling queues**
  – **Ready queue**
    • ready to be executed processes
  – **Device queues**
    • processes waiting for an I/O device
  – Processes **migrate** among the various queues
Ready Queue and I/O Device Queues
Process Scheduling: Queuing Representation

- Ready queue
- I/O request
- I/O queue
- Time slice expired
- Fork a child
- Wait for an interrupt
- Child executes
- Interrupt occurs
Context Switching

- Suspend the current process and resume a next one from its last suspended state.
Context Switching

• Overhead
  – Save and restore CPU states
  – Warm up instruction and data cache
    • Cache data of previous process is not useful for new process

• In Linux 3.6.0 on an Intel Xeon 2.8Ghz
  – About 1.8 us
  – ~ 5040 CPU cycles
  – ~ thousands of instructions
Process Creation

• **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

• Generally, process identified and managed via a **process identifier (pid)**
A Process Tree in Linux

init
  pid = 1

login
  pid = 8415

kthread
  pid = 2

bash
  pid = 8416

ps
  pid = 9298

emacs
  pid = 9204

khelper
  pid = 6

pdflush
  pid = 200

sshd
  pid = 3028

sshd
  pid = 3610

tcsch
  pid = 4005
‘pstree’ output
Process Creation

- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process’ memory space with a new program
Example: Forking a Process in UNIX

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```
Example: Forking a Process in Windows

```c
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
                        "C:\WINDOWS\system32\mspaint.exe", /* command */
                        NULL, /* don’t inherit process handle */
                        NULL, /* don’t inherit thread handle */
                        FALSE, /* disable handle inheritance */
                        0, /* no creation flags */
                        NULL, /* use parent’s environment block */
                        NULL, /* use parent’s existing directory */
                        &si,
                        &pi))
    {
        printf(stderr, "Create Process Failed");
        return -1;
    }

    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```
Process Termination

- Normal termination via `exit()` system call.
  - Exit by itself.
  - Returns status data from child to parent (via `wait()`)
  - Process’s resources are deallocated by operating system
- Forced termination via `kill()` system call
  - Kill someone else (child)
- **Zombie** process
  - If no parent waiting (did not invoke `wait()`)  
- **Orphan** process
  - If parent terminated without invoking `wait`
  - Q: who will be the parent of a orphan process?
  - A: Init process