Process

Lecture 4

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

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

• Today
  – Process concept
  – Context switching
Process

• Process: a program in execution
  – 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 class
Virtual Memory vs. Physical Memory

Virtual Memory

Physical Memory

Process A

Process B

Process C
Process State

- **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
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 *)

```bash
(*) # cat /sys/kernel/slab/task_struct/object_size
```

(currently executing process)
Context Switch

- **Process $P_0$**: Executing
  - Interrupt or system call
  - Save state into PCB$_0$
  - Reload state from PCB$_1$
- **Operating System**: Idle
- **Process $P_1$**: Executing
  - Interrupt or system call
  - Save state into PCB$_1$
  - Reload state from PCB$_0$
Context Switch Overhead

• Context switching
  – 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 Intel Xeon 2.8Ghz
  – About 1.8 us
  – ~ 5040 CPU cycles
  – ~ thousands of instructions
Process Scheduling

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

• Decides which process to run next
  – Among ready processes

• Maintains scheduling queues of processes
  – 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
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
‘pstree’ output
Process Creation

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- 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 */
        execlp("/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))
    {
        fprintf(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);
}
Example: Forking a Process in UNIX

```c
int count = 0;
int main()
{
    int pid = fork();
    if (pid == 0){
        count++;
        printf("Child: %d\n", count);
    } else{
        wait(NULL);
        count++;
        printf("Parent: %d\n", count);
    }
    count++;
    printf("Main: %d\n", count);
    return 0;
}
```

- **Hints**
  - Each process has its own private address space
  - `Wait()` blocks until the child finish

- **Output?**
  - Child: 1
  - Main: 2
  - Parent: 1
  - Main: 2
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?