Processes

Introduction to Computer Systems
Nov. 26, 2015

Instructors:
Dohyung Kim
Processes

- **Definition:** A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- **Process provides each program with two key abstractions:**
  - *Logical control flow*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
Processes

Definition: A process is an instance of a running program.
- One of the most profound ideas in computer science
- Not the same as “program” or “processor”

Process provides each program with two key abstractions:
- **Logical control flow**
  - Each program seems to have exclusive use of the CPU
  - Provided by kernel mechanism called *context switching*
- **Private address space**
  - Each program seems to have exclusive use of main memory.
  - Provided by kernel mechanism called *virtual memory*
Private Address Spaces

- Each process has its own private address space
Multiprocessing: The Illusion

- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, ...
  - Background tasks
    - Monitoring network & I/O devices
Multiprocessing Example

- Running program “top” on Mac
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)
Multiprocessing: The (Traditional) Reality

- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for nonexecuting processes saved in memory
Multiprocessing: The (Traditional) Reality

- Save current registers in memory

Diagram:
- Memory
  - Stack
  - Heap
  - Data
  - Code
  - Saved registers

- CPU
  - Registers
Multiprocessing: The (Traditional) Reality

- Schedule next process for execution
Multiprocessing: The (Traditional) Reality

- Load saved registers and switch address space (context switch)
Multiprocessing: The (Modern) Reality

- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel
Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent*) if their flows overlap in time.
- Otherwise, they are *sequential*.
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C

![Process Timeline Diagram]
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other
Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.

- Control flow passes from one process to another via a *context switch*
System Call Error Handling

- On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.
- Hard and fast rule:
  - You must check the return status of every system-level function
  - Only exception is the handful of functions that return `void`
- Example:

```c
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```
Error-reporting functions

- Can simplify somewhat using an *error-reporting function*:

```c
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
```

```c
if ((pid = fork()) < 0)
    unix_error("fork error");
```
Error-handling Wrappers

- We simplify the code we present to you even further by using Stevens-style error-handling wrappers:

```c
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

```c
pid = Fork();
```
Obtaining Process IDs

- `pid_t getpid(void)`
  - Returns PID of current process

- `pid_t getppid(void)`
  - Returns PID of parent process
Creating and Terminating Processes

From a programmer’s perspective, we can think of a process as being in one of three states

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel

- **Stopped**
  - Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)

- **Terminated**
  - Process is stopped permanently
Process State Transition

- **new** → admitted
- **ready** → interrupt
- **running** → exit
- **waiting** → scheduler dispatch
- **waiting** → I/O or event completion
- **waiting** → I/O or event wait
- **ready** → scheduler dispatch
Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

- **`void exit(int status)`**
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- **`exit` is called once but never returns.**
Terminating Process

- `atexit()` registers functions to be executed upon exit

```c
void cleanup(void)
{
    printf("cleaning up", --x);
    exit(0);
}

void fork6()
{
    atexit (cleanup);
    fork();
    exit(0);
}
```
Creating Processes

- **Parent process** creates a new running **child process** by calling **fork**

- **int fork(void)**
  - Returns 0 to the child process, child’s PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent’s virtual address space.
    - Child gets identical copies of the parent’s open file descriptors
    - Child has a different PID than the parent

- **fork** is interesting (and often confusing) because it is called *once* but returns *twice*
# fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

- Call once, return twice
- Concurrent execution
  - Can’t predict execution order of parent and child
- Duplicate but separate address space
  - `x` has a value of 1 when fork returns in parent and child
  - Subsequent changes to `x` are independent
- Shared open files
  - `stdout` is the same in both parent and child

```bash
linux> ./fork
parent: x=0
cchild : x=2
```
fork Example

```c
int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    } /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}

/* Parent */
printf("parent: x=%d\n", --x);
exit(0);
}

fork.c
```
fork Example: Two consecutive forks

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

Feasible output:
L0
L1
Bye
Bye
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye
Bye
fork Example: Nested forks in parent

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:
- L0
- L1
- Bye
- L2
- Bye

Infeasible output:
- L0
- Bye
- L1
- Bye
- L2
**fork Example: Nested forks in children**

```c
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

Feasible output:
```
L0
Bye
L1
L2
Bye
Bye
```

Infeasible output:
```
L0
Bye
L1
Bye
L1
Bye
Bye
L2
```
Reaping Child Processes

■ Idea
  ▪ When process terminates, it still consumes system resources
    ▪ Examples: Exit status, various OS tables
  ▪ Called a “zombie”
    ▪ Living corpse, half alive and half dead

■ Reaping
  ▪ Performed by parent on terminated child (using `wait` or `waitpid`)
  ▪ Parent is given exit status information of zombie child process
  ▪ Kernel then deletes zombie child process

■ What if parent doesn’t reap?
  ▪ If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
  ▪ So, only need explicit reaping in long-running processes
    ▪ e.g., shells and servers
Zombie Example

void fork7() {
    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 */
    }
}

Zombie Example

```
ps shows child process as “defunct” (i.e., a zombie)
Killing parent allows child to be reaped by init
```
Non-terminating Child Example

Child process still active even though parent has terminated

Must kill child explicitly, or else will keep running indefinitely
wait: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the `pid` of the child process that terminated
  - If `child_status != NULL`, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - See textbook for details
```c
void fork9() {
    int child_status;
    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

Feasible output:
- HC
- HP
- CT
- Bye

Infeasible output:
- HP
- CT
- Bye
- HC
Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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 terminate abnormally\n", wpid);
    }
}
```

forks.c
waitpid: Waiting for a Specific Process

- **`pid_t waitpid(pid_t pid, int &status, int options)`**
  - Suspends current process until specific process terminates
  - Various options: WIFEXITED, WEXITSTATUS (see textbook)

```c
void fork11() {
    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 terminate abnormally\n", wpid);
} 
```
execve: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`

- **Loads and runs in the current process:**
  - Executable file `filename`
    - Can be object file or script file beginning with `#!` interpreter (e.g., `#!/bin/bash`)
  - ...with argument list `argv`
    - By convention `argv[0] == filename`
  - ...and environment variable list `envp`
    - “name=value” strings (e.g., `USER=droh`)
      - `getenv`, `putenv`, `printenv`

- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context

- **Called once and never returns**
  - ...except if there is an error
Structure of the stack when a new program starts
**execve Example**

- **Executes** "`/bin/ls -lt /usr/include`" in child process using current environment:

  ```c
  if ((pid = Fork()) == 0) {
    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
      printf("%s: Command not found.\n", myargv[0]);
      exit(1);
    }
  }
  ```

  ```
  Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
  ```
Summary

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on a single core, though
  - Each process appears to have total control of processor + private memory space

- **Spawning processes**
  - `fork` : One call, two returns

- **Process completion**
  - `exit` : One call, no return

- **Reaping and waiting for processes**
  - `wait` or `waitpid`

- **Loading and running programs**
  - `execve` : One call, (normally) no return