Processes
CSE 410 Winter 2017

Instructor: Justin Hsia
Teaching Assistants: Kathryn Chan, Kevin Bi, Ryan Wong, Waylon Huang, Xinyu Sui
Administrivia

- Homework 4 due Tuesday (2/28)
- Lab 4 due next Tuesday (3/7)

**Final Exam:** Tuesday, March 14 @ 2:30pm (MGH 241)
- Review Session: Sunday, March 12 @ 1:30pm in SAV 264
- Cumulative (midterm clobber policy applies)
- TWO double-sided handwritten 8.5×11” cheat sheets
  - Recommended that you reuse or remake your midterm cheat sheet
Processes

- Processes and context switching
- Creating new processes
  - `fork()`, `exec*()`, and `wait()`
- Zombies
What is a process?

It’s an illusion!
What is a process?

- Another *abstraction* in our computer system
  - Provided by the OS
  - OS uses a data structure to represent each process
  - Maintains the *interface* between the program and the underlying hardware (CPU + memory)

- What do *processes* have to do with *exceptional control flow*?
  - Exceptional control flow is the *mechanism* the OS uses to enable *multiple processes* to run on the same system

- What is the difference between:
  - A processor? A program? A process?
Processes

- 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*
What is a process?

It’s an illusion!
What is a process?

It’s an illusion!
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: The Reality

- Single processor executes multiple processes **concurrently**
  - Process executions interleaved, CPU runs *one at a time*
  - Address spaces managed by virtual memory system (later in course)
  - *Execution context* (register values, stack, ...) for other processes saved in memory
Multiprocessing

- Context switch
  1) Save current registers in memory
Multiprocessing

- **Context switch**
  1) Save current registers in memory
  2) **Schedule next process for execution**
Multiprocessing

- **Context switch**
  1) Save current registers in memory
  2) Schedule next process for execution
  3) Load saved registers and switch address space
Multiprocessing: The (Modern) Reality

- **Multicore processors**
  - Multiple CPUs (“cores”) on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Kernel schedules processes to cores
    - *Still constantly swapping processes*
Concurrent Processes

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

Assume only one CPU.
User’s View of Concurrency

- Control flows for concurrent processes are physically disjoint in time
  - CPU only executes instructions for one process at a time

- However, the user can *think of* concurrent processes as executing at the same time, in *parallel*
Context Switching

- Processes are managed by a *shared* chunk of OS code called the **kernel**
  - The kernel is not a separate process, but rather runs as part of a user process

- In x86-64 Linux:
  - Same address in each process refers to same shared memory location

Assume only one CPU
Context Switching

- Processes are managed by a *shared* chunk of OS code called the **kernel**
  - The kernel is not a separate process, but rather runs as part of a user process
- Context switch passes control flow from one process to another and is performed using kernel code
Processes

- Processes and context switching
- **Creating new processes**
  - `fork()`, `exec*()`, and `wait()`
- Zombies
Creating New Processes & Programs

Process 1

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

Process 2

“Memory”
- Stack
- Heap
- Data
- Code

“CPU”
- Registers

fork()

exec*()

Chrome.exe
Creating New Processes & Programs

- fork-exec model (Linux):
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process’ code and address space with the code for a different program
    - Family: execv, execl, execve, execle, execvp, execlp
  - `fork()` and `execve()` are system calls

- Other system calls for process management:
  - getpid()
  - exit()
  - wait(), waitpid()
**fork**: Creating New Processes

- `pid_t fork(void)`
  - Creates a new “child” process that is *identical* to the calling “parent” process, including all state (memory, registers, etc.)
  - Returns 0 to the child process
  - Returns child’s process ID (PID) to the parent process

- Child is *almost* identical to parent:
  - Child gets an identical (but separate) copy of the parent’s virtual address space
  - Child has a different PID than the parent

- `fork` is unique (and often confusing) because it is called *once* but returns “*twice*”

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding fork

**Process X (parent)**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

**Process Y (child)**

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding fork

Process X  (parent)

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Process Y  (child)

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```
Understanding fork

Process X (parent)

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Process Y (child)

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Which one appears first?

Hello from parent

Hello from child
Fork Example

```c
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

- Both processes continue/start execution after `fork`
  - Child starts at instruction after the call to `fork` (storing into `pid`)
- Can’t predict execution order of parent and child
- Both processes start with `x=1`
  - Subsequent changes to `x` are independent
- Shared open files: stdout is the same in both parent and child
Modeling `fork` with Process Graphs

- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program
  - Each vertex is the execution of a statement
  - \( a \rightarrow b \) means \( a \) happens before \( b \)
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges

- Any *topological sort* of the graph corresponds to a feasible total ordering
  - Total ordering of vertices where all edges point from left to right
Fork Example: Possible Output

```c
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

---

- **x** = 1
  - Fork
  - **x** = 2
    - **Child**
      - ++x
        - printf
      - Bye
    - printf
  - Bye
- **x** = 0
  - **Parent**
    - --x
      - printf
    - Bye
  - printf
Peer Instruction Question

- Are the following sequences of outputs possible?

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

Seq 1:  
L0  L1  Bye  Bye  L2  Bye  L2
Seq 2:  
L0  Bye  L1  Bye  L1  Bye  Bye

A. No  No  
B. No  Yes  
C. Yes  No  
D. Yes  Yes  
E. We’re lost...
Fork-Exec

- fork-exec model:
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process’ code and address space with the code for a different program
  - Whole family of `exec` calls – see `exec(3)` and `execve(2)`

```c
void fork_exec(char *path, char *argv[]) {  
    pid_t pid = fork();
    if (pid != 0) {
        printf("Parent: created a child %d\n", pid);
    } else {
        printf("Child: about to exec a new program\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```

*Note: the return values of `fork` and `exec*` should be checked for errors*
Exec-ing a new program

Very high-level diagram of what happens when you run the command "ls" in a Linux shell:

This is the loading part of CALL!
execve Example

Execute “/usr/bin/ls –l lab4” in child process using current environment:

```
myargv[argc] = NULL
myargv[2]
myargv[1]
myargv[0]
```

```
envp[n] = NULL
envp[n-1]
...
envp[0]
```

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

Run the `printenv` command in a Linux shell to see your own environment variables
Structure of the Stack when a new program starts

- Null-terminated environment variable strings
  - envp[n] == NULL
  - envp[n-1]
  - ...
  - envp[0]

- Null-terminated command-line arg strings
  - argv[argc] = NULL
  - argv[argc-1]
  - ...
  - argv[0]

- Stack frame for libc_start_main
- Future stack frame for main

Bottom of stack
- environ (global var)
- envp (in %rdx)

Top of stack
- argv (in %rsi)
- argc (in %rdi)
exit: Ending a process

- **void exit(int status)**
  - Exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit
Processes

- Processes and context switching
- Creating new processes
  - `fork()`, `exec*()`, and `wait()`
- Zombies
Zombies

- When a process terminates, it still consumes system resources
  - Various tables maintained by OS
  - Called a “zombie” (a living corpse, half alive and half dead)

- Reaping is performed by parent on terminated child
  - Parent is given exit status information and 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)
    - Note: on more recent Linux systems, init has been renamed systemd
  - In long-running processes (e.g. shells, servers) we need explicit reaping
wait: Synchronizing with Children

- **int wait(int **child_status)**
  - Suspends current process (i.e. the parent) until one of its children terminates
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If `child_status` != NULL, then the `*child_status` value indicates why the child process terminated
    - Special macros for interpreting this status – see `man wait(2)`

- **Note:** If parent process has multiple children, `wait` will return when *any* of the children terminates
  - `waitpid` can be used to wait on a specific child process
wait: Synchronizing with Children

```c
void fork_wait() {
    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  Bye

Infeasible output: 
HP  CT  Bye  HC
Example: Zombie

```c
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 */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
```

```
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 */
    }
}
```

```
ps shows child process as “defunct”
Killing parent allows child to be reaped by init
```

```
linux> ps
    PID TTY          TIME CMD
6585 tttyp9    00:00:00 tcsh
6639 tttyp9    00:00:03 forks
6640 tttyp9    00:00:00 forks <defunct>
6641 tttyp9    00:00:00 ps
```

```
linux> kill 6639
[1]    Terminated
```

```
linux> ps
    PID TTY          TIME CMD
6585 tttyp9    00:00:00 tcsh
6642 tttyp9    00:00:00 ps
```
Example: Non-terminating Child

```c
void fork8() {
    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);
    }
}
```

```
void fork8() {
    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);
    }
}
```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely
Process Management Summary

- **fork** makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- **exec** replaces current process from file (new program)
  - Two-process program:
    - First **fork**()
    - if (pid == 0) { /* child code */ } else { /* parent code */ }
  - Two different programs:
    - First **fork**()
    - if (pid == 0) { execv(...) } else { /* parent code */ }

- **wait** or **waitpid** used to synchronize parent/child execution and to reap child process
Summary

❖ Processes
  ▪ At any given time, system has multiple active processes
  ▪ On a one-CPU system, only one can execute at a time, but each process appears to have total control of the processor
  ▪ OS periodically “context switches” between active processes
    • Implemented using *exceptional control flow*

❖ Process management
  ▪ `fork`: one call, two returns
  ▪ `execve`: one call, usually no return
  ▪ `wait` or `waitpid`: synchronization
  ▪ `exit`: one call, no return
Detailed examples:

- Consecutive forks
- `wait()` example
- `waitpid()` example
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
L1
Bye
Bye
Bye
Bye
Example: Three consecutive forks

- Both parent and child can continue forking

```c
void fork3() {
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```
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;
    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);
    }
}
```
waitpid(): Waiting for a Specific Process

```
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 = 0; i < N; 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);
    }
}
```