COMP 4511

Process

Goal

- Overview of Process
- Hands-on practices via POSIX
- Lab Exercise: Create a simple shell with child process creation

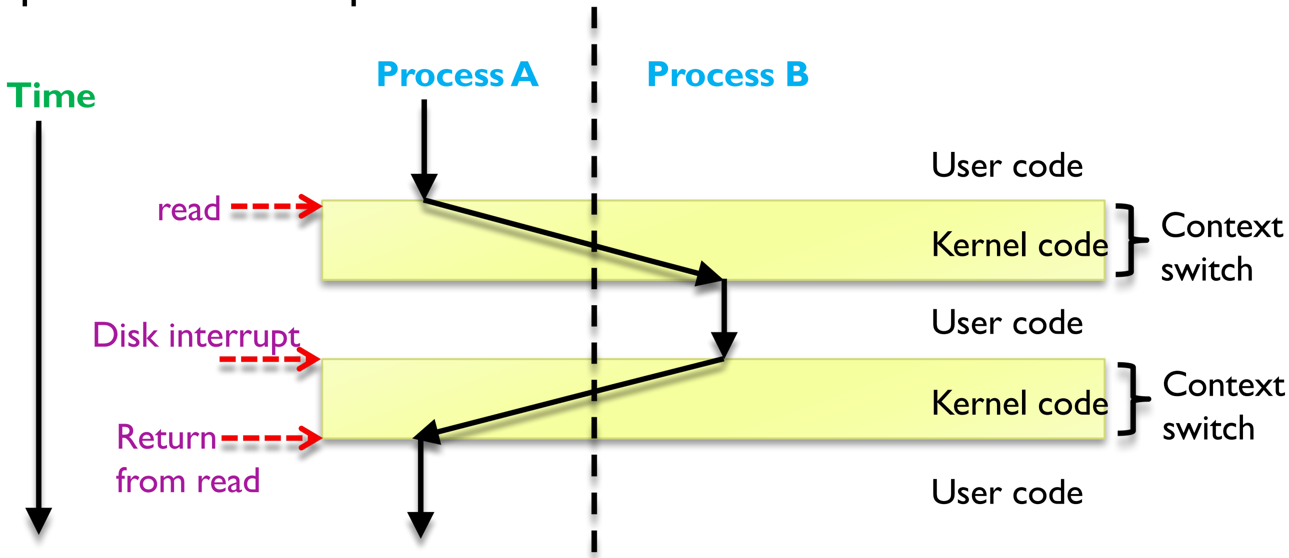
Overview of Process

What is a process?

- Definition: an executable instance of a program
  - A process is the context maintained for an executing problem
  - Process is different from a program
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the processor
  - Private virtual address
    - Each program seems to have exclusive use of main memory
- How are these illusions maintained?
  - Processes executions interleaved (multitasking) or run on separate cores (parallel)
  - Private address spaces managed by virtual memory system

Context switching

- Processes are managed by the kernel
- Control passes from one process to another via a context switch



What makes up a process?

- Program code
- Machine registers
- Global data
- Stack
- Open files
- An environment

Process context

- Process ID (`pid`)
- Parent process ID (`ppid`)
- Current directory
- File descriptor table
- Environment
- Pointer to program code
- Pointer to data (memory for global variables)
- Pointer to stack (memory for local variables)
- Pointer to heap (dynamically allocated memory)
- Execution priority
- Signal information

```
pid_t myid = getpid();
pid_t myparentid = getppid();
```

Linux processes

- Virtual address space
  - The virtual address space is the memory that contains the code to execute as well as the process stack and data
- Process descriptor (`struct task_struct`): data structure in the kernel to keep track of that process
  - Virtual address space map
  - Current status of the process
  - Execution priority of the process
  - Resource usage of the process
  - Current signal mask
  - Owner of the process

Hands-on practices via POSIX

What is POSIX?

- An acronym for Portable Operating System Interface
- POSIX defines the application programming interface or software compatibility with Unix-like operating systems and other operating systems
- On Unix-like systems, **`unistd.h`** provides access to the POSIX for C/C++ programming
  - It provides process-related function calls such as `fork(), pipe(),…`
  - They are wrapper functions for the related system calls in Linux kernel such as `sys_fork(), sys_pipe(), …`
  - Reference: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/unistd.h.html

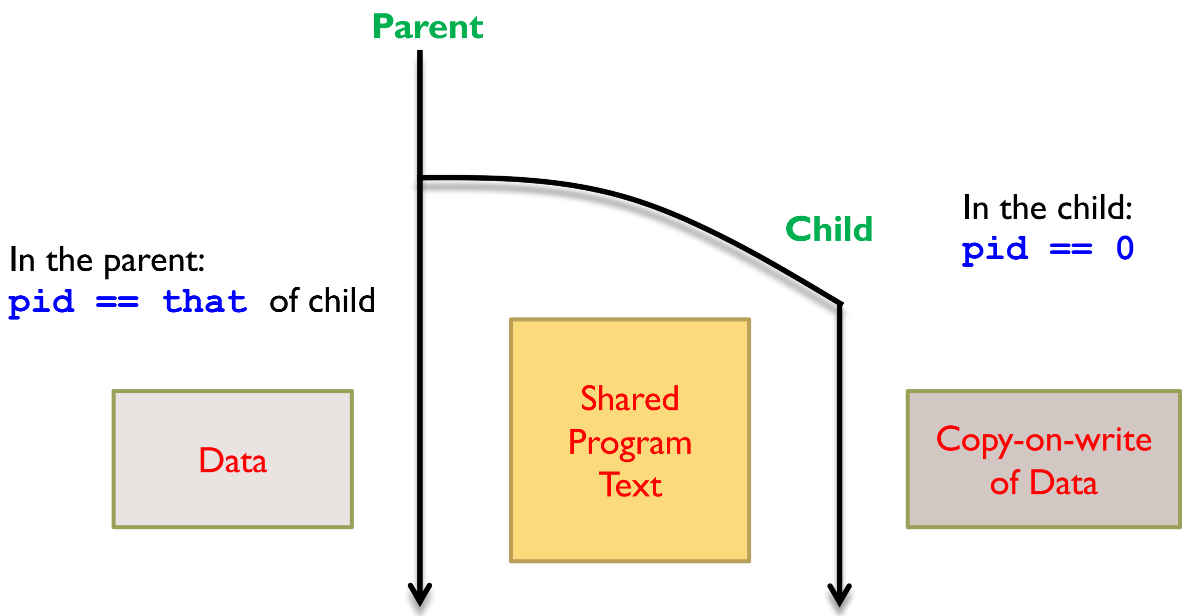
Creating a process: fork

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

- Create a child process
  - The child is an (almost) exact copy of the parent
  - The new process and the old process both continue in parallel from the statement that follows the `fork()`
- Returns
  - To child:
    - 0 on success
  - To parent:
    - Process ID of the child process
    - -1 on error, sets `errno`

The return PID is important for both parent and child processes

Creating a process: fork()



`fork()` is called once, but returns twice!

What is fork() good for?

- Two processes run concurrently if their flows overlap in time
- What does concurrency gain us?
  - The appearance that multi-actions are occurring at the same time
  - If done right, your program can improve throughput
- `fork()` creates a new process that runs concurrently
- Why concurrency?
  - Exploit natural concurrent structure of an application
    - Easier to program multiple independent and concurrent activities
  - Better resource utilization
    - Resource unused by one application can be used by others
  - Better average response time
    - No need to wait for other applications to make progress

fork HelloWorld - fork_hello.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
 int main() {
   pid_t pid;
   int i;
   /* Create a new process by
duplicating the calling process */
   pid = fork();

    if ( pid > 0 ) {
              /* parent process */
       for (i=0; i<10; i++)
           printf("Parent %d\n",i);
   } else { /* child process */
       for (i=0; i<10; i++)
           printf("Child %d\n",i);
   }
   return 0;
}
```

```
cspeter — cspeter@ubuntu: ~/comp4511_labs — ssh -p 3022 cspeter@localhost
cspeter@ubuntu:~/comp4511_labs$ gcc fork_hello.c
cspeter@ubuntu:~/comp4511_labs$ ./a.out
Parent 0
Parent 1
Parent 2
Parent 3
Parent 4
Parent 5
Parent 6
Parent 7
Parent 8
Parent 9
cspeter@ubuntu:~/comp4511_labs$ Child 0
Child 1
Child 2
Child 3
Child 4
Child 5
Child 6
Child 7
Child 8
Child 9
```

The parent process terminated here…

The child process becomes a zombie process here because the parent is already terminated and it won't be able to return to its parent (Press Control-C to terminate it)

Zombies

- What happens on termination?
  - When process terminates, still consumes system resources
  - Entries in various table & information maintained by OS
- Called a "zombie", waiting parent to reap it
- What if parent does not reap?
  - If any parent terminates without reaping a child, then child will be reaped by `init` process

A better fork example – fork_with_wait.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
  pid_t pid;
  int i;
  /* Create a new process by
  duplicating the calling process */
  pid = fork();
  if ( pid > 0 ) { /* parent process */
     for (i=0; i<10; i++)
        printf("Parent %d\n",i);
     wait(0);
    /* Wait for the child process */
  } else { /* child process */
     for (i=0; i<10; i++)
         printf("Child %d\n",i);
  }
  return 0;
}
```

```
cspeter@ubuntu:~/comp4511_labs$ gcc fork_with_wait.c
cspeter@ubuntu:~/comp4511_labs$ ./a.out
Parent 0
Parent 1
Parent 2
Parent 3
Parent 4
Parent 5
Parent 6
Parent 7
Parent 8
Parent 9
Child 0
Child 1
Child 2
Child 3
Child 4
Child 5
Child 6
Child 7
Child 8
Child 9
cspeter@ubuntu:~/comp4511_labs$
```

The parent process finished its print job and waited

The child process returned and the parent process terminated

Waiting a child process: wait, waitpid, waitid

```
#include <sys/types.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int
options);
```

- All of these system calls are used to wait for state changes in a child of the calling process
- status is the pointer pointing to an int that captures the return status. You can input 0 (NULL) if no return status is needed
- Reference: http://linux.die.net/man/2/wait OR man wait

Checking return status – fork_wait_status.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main() {
   int child_status;
   pid_t child_pid ;
   pid_t pid = fork();
   if ( pid == 0 ) { /* child */
      return 0; }
   else { /* parent */
      printf("Parent PID %d\n", getpid());
      child_pid = wait(&child_status);
      printf("Child PID %d with status %d\n”
       , child_pid, child_status);
   }
   return 0;
}
```

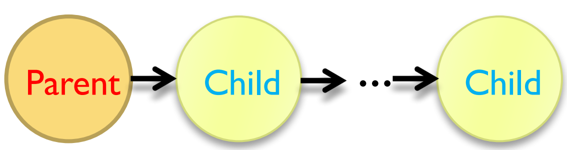
```
cspeter — cspeter@ubuntu: ~/comp4511_labs — ssh -p 3022 cspeter@localhost -
cspeter@ubuntu:~/comp4511_labs$ gcc fork_wait_status.c
cspeter@ubuntu:~/comp4511_labs$ ./a.out
Parent PID 1719
Child PID 1720 with status 0
cspeter@ubuntu:~/comp4511_labs$
```

Status number 0 means exit without any error

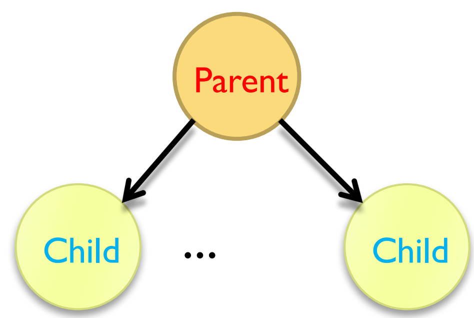
Note: If parent has multiple children, wait will return when one of them (order not known!) completes.

Child process creation pattern: Chain and fan

Chain



Fan

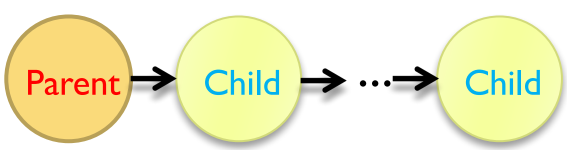


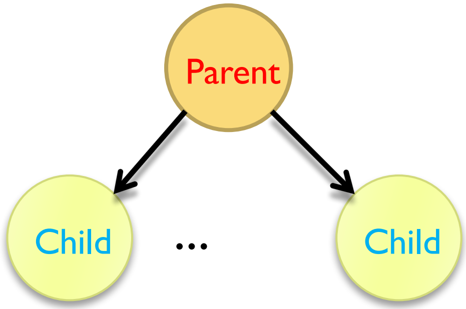
Child process creation pattern: Chain and fan

Chain

Fan

```
pid_t childpid;
for (i = 1;i < n; ++i)
    if (childpid = fork())
        break;
```

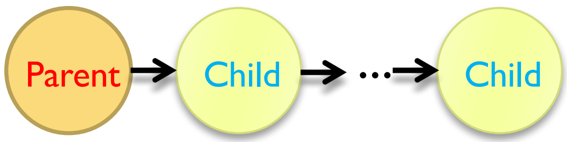




Child process creation pattern: Chain and fan

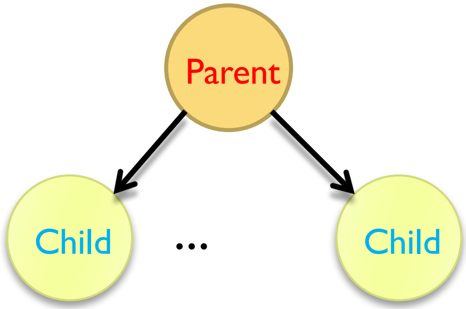
Chain

```
pid_t childpid;
for (i = 1;i < n; ++i)
    if (childpid = fork())
        break;
```



Fan

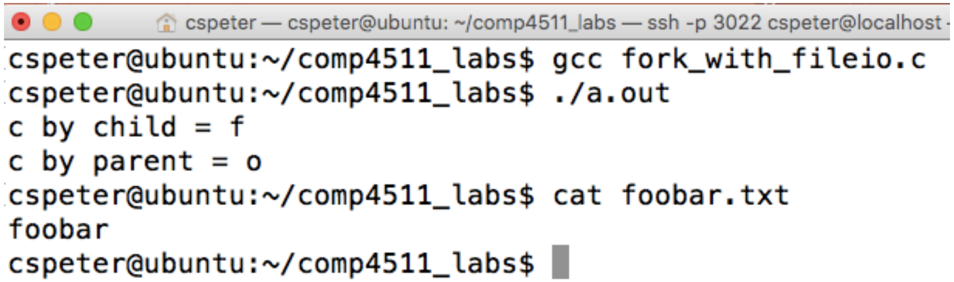
```
pid_t childpid;
for (i = 1;i < n; ++i)
    if ((childpid = fork()) <= 0)
        break;
```



fork with fileI/O – child inherits open files

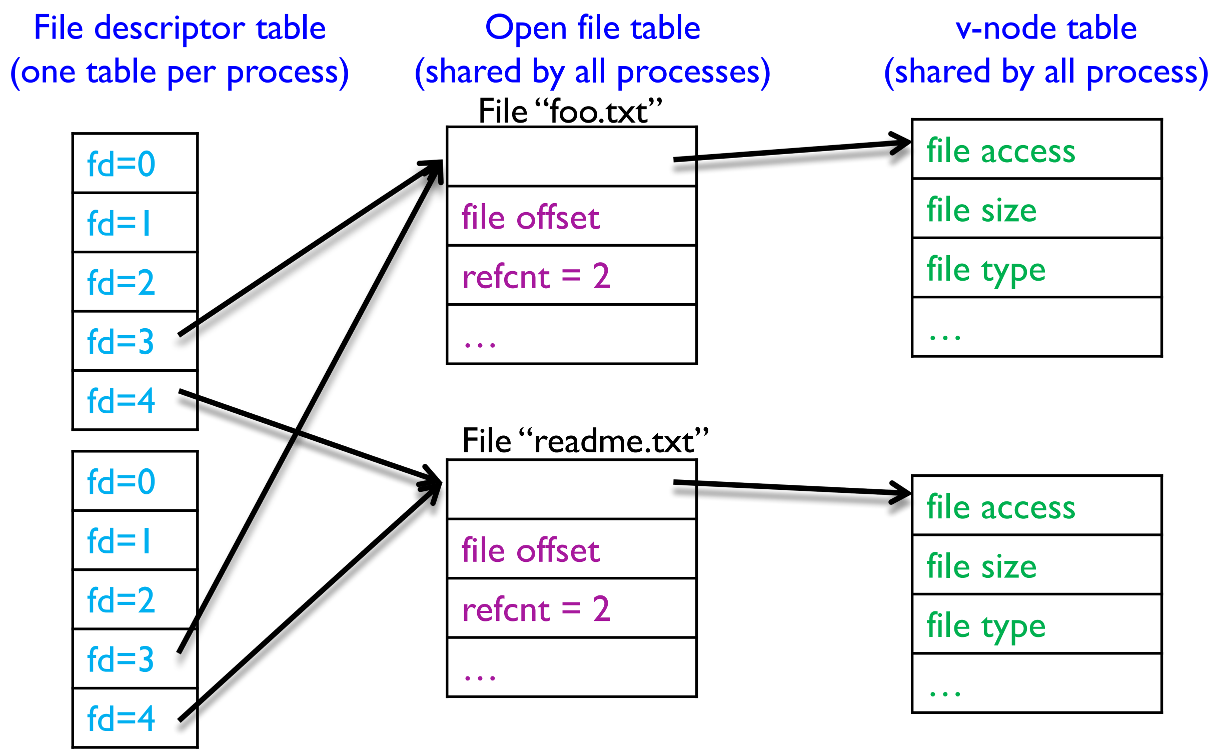
- The output of the program

```c
#include <stdio.h>
#include <unistd.h> /* POSIX header */
#include <fcntl.h> /* open for POSIX */
#include <sys/types.h>
/* foobar.txt: A text file contains 6 characters: foobar */
int main() {
  char c;
  int fd = open("foobar.txt", O_RDONLY,0);
  pid_t pid = fork();
  if ( pid == 0 ) { /* child process */
  read(fd, &c, 1); /* read a char */
  printf("c by child = %c\n", c);
  return 0; /* terminate */
 }
 wait(0); /* wait for the child process */
 read(fd, &c, 1);
 printf("c by parent = %c\n", c);
 close(fd);
 return 0;
}
```


cspeter — cspeter@ubuntu: ~/comp4511_labs — ssh -p 3022 cspeter@localhost

```
cspeter@ubuntu:~/comp4511_labs$ gcc fork_with_fileio.c
cspeter@ubuntu:~/comp4511_labs$ ./a.out
c by child = f
c by parent = o
cspeter@ubuntu:~/comp4511_labs$ cat foobar.txt
foobar
cspeter@ubuntu:~/comp4511_labs$
```

Child process inherits open files



When a process terminates

- When a child process terminates:
  - Open files are flushed and closed
  - Child's resources are de-allocated
    - File descriptors, memory, semaphores, file locks, …
- Parent process is notified via signal SIGCHLD
- Exit status is available to parent via `wait()`

Voluntary termination	Involuntary termination
Normal exit: exit(0)	Fatal error: divide by 0, core dump, segment fault
Error exit: exit(1)	Killed by another process: kill()

Sleep - sleep the current process for the specified number of seconds

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

int main() {
    printf("Start to sleep for 10s\n");
    sleep(10);
    printf("End of sleep\n");
return 0;
}
```

```
cspeter@ubuntu:~/comp4511_labs$ gcc sleep_example.c
cspeter@ubuntu:~/comp4511_labs$ ./a.out
Start to sleep for 10s
End of sleep
cspeter@ubuntu:~/comp4511_labs$ █
```

Lab Exercise: Create a simple shell with child process creation

- `myshell_skeleton.c` is provided as the starting point
- Complete the `process_cmd` function
- Make sure that you use the wait function to reap the child process
- Commands supported:
  - exit
  - child [n]
    - Create a child process that runs for n seconds (using sleep)
    - Print out the PID of the child process
    - Print out the status code when the child process terminates

```
void process_cmd(char *cmdline)
{
        // printf("%s\n", cmdline);

}
```

cspeter — cspeter@ubuntu: ~/comp4511_labs — ssh -p 3022

```
cspeter@ubuntu:~/comp4511_labs$ gcc -o myshell myshell.c
cspeter@ubuntu:~/comp4511_labs$ ./myshell
myshell> child 5
child pid 1955 is started
child pid 1955 is terminated with status 0
myshell> child 2
child pid 1956 is started
child pid 1956 is terminated with status 0
myshell> exit
myshell is terminated with pid 1954
cspeter@ubuntu:~/comp4511_labs$
```