Processes
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Midterms Graded
  - If you did not receive an email from Gradescope let us know
- Homework 4 coming soon!
  - Cache questions
- Lab 4 coming soon!
  - Cache runtimes and parameter puzzles
Roadmap

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:

```assembly
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
1000110100000100
1000100111000010
11000011111101000011111
```

OS:

Virtual memory
Memory allocation
Java vs. C

Memory & data organization:
Integers & floats
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
Control Flow

- **So far:** we’ve seen how the flow of control changes as a *single program* executes

- **Reality:** multiple programs running *concurrently*
  - How does control flow across the many components of the system?
  - In particular: More programs running than CPUs

- *Exceptional control flow* is basic mechanism used for:
  - Transferring control between *processes* and OS
  - Handling *I/O* and *virtual memory* within the OS
  - Implementing multi-process apps like shells and web servers
  - Implementing concurrency
Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s control flow (or flow of control)

\[
\begin{align*}
\text{<startup>} & \quad \text{instr}_1 \\
\quad & \quad \text{instr}_2 \\
\quad & \quad \text{instr}_3 \\
\quad & \quad \ldots \\
\quad & \quad \text{instr}_n \\
\text{<shutdown>} &
\end{align*}
\]

Physical control flow
Altering the Control Flow

- Up to now, two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  - Both react to changes in *program state*

- Processor also needs to react to changes in *system state*
  - Unix/Linux user hits “Ctrl-C” at the keyboard
  - User clicks on a different application’s window on the screen
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - System timer expires

- Can jumps and procedure calls achieve this?
  - No – the system needs mechanisms for “*exceptional*” control flow!
Java Digression

- Java has exceptions, but they’re *something different*
  - **Examples:** NullPointerException, MyBadThingHappenedException, ...
  - *throw* statements
  - *try/catch* statements (“throw to youngest matching catch on the call-stack, or exit-with-stack-trace if none”)

- Java exceptions are for reacting to (unexpected) program state
  - Can be implemented with stack operations and conditional jumps
  - A mechanism for “many call-stack returns at once”
  - Requires additions to the calling convention, but we already have the CPU features we need

- System-state changes on previous slide are mostly of a different sort (asynchronous/external except for divide-by-zero) and implemented very differently
Exceptional Control Flow

- Exists at all levels of a computer system

- Low level mechanisms
  - **Exceptions**
    - Change in processor’s control flow in response to a system event (i.e. change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software

- Higher level mechanisms
  - **Process context switch**
    - Implemented by OS software and hardware timer
  - **Signals**
    - Implemented by OS software
    - Covered in CSE 451 and CSE 466
Exceptions

- An exception is a transfer of control to the operating system (OS) kernel in response to some event (i.e., change in processor state).
  - Kernel is the memory-resident part of the OS.
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C.

How does the system know where to jump to in the OS?
Exception Table

- A jump table for exceptions (also called *Interrupt Vector Table*)
  - Each type of event has a unique exception number $k$
  - $k = \text{index into exception table (a.k.a interrupt vector)}$
  - Handler $k$ is called each time exception $k$ occurs
# Exception Table (Excerpt)

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>
Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
**Asynchronous Exceptions (Interrupts)**

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to “next” instruction

- **Examples:**
  - I/O interrupts
    - Hitting Ctrl-C on the keyboard
    - Clicking a mouse button or tapping a touchscreen
    - Arrival of a packet from a network
    - Arrival of data from a disk
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - **Intentional**: transfer control to OS to perform some function
    - **Examples**: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction ("current" instr did what it was supposed to)
  - **Faults**
    - **Unintentional** but possibly recoverable
    - **Examples**: page faults, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - **Aborts**
    - **Unintentional** and unrecoverable
    - **Examples**: parity error, machine check (hardware failure detected)
    - Aborts current program
System Calls

- Each system call has a unique ID number
- Examples for Linux on x86-64:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
<td>Read file</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>Write file</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
<td>Open file</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>Close file</td>
</tr>
<tr>
<td>4</td>
<td>stat</td>
<td>Get info about file</td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
<td>Create process</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
<td>Execute a program</td>
</tr>
<tr>
<td>60</td>
<td>_exit</td>
<td>Terminate process</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
<td>Send signal to process</td>
</tr>
</tbody>
</table>
Traps Example: Opening File

- User calls `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
  ...
  e5d79: b8 02 00 00 00           mov $0x2,%eax # open is syscall 2
  e5d7e: 0f 05                      syscall # return value in %rax
  e5d80: 48 3d 01 f0 ff ff          cmp $0xfffffffffffff001,%rax
  ...
  e5dfa: c3                         retq
```

- `%rax` contains syscall number
- Other arguments in `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

```
int a[1000];
int main ()
{
    a[500] = 13;
}
```

```
80483b7: c7 05 10 9d 04 08 0d movl $0xd,0x8049d10
```

- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try
Fault Example: Invalid Memory Reference

```c
int a[1000];
int main()
{
    a[5000] = 13;
}
```

User Process

```assembly
80483b7: c7 05 60 e3 04 08 0d movl $0xd, 0x804e360
```

OS

- Page fault handler detects invalid address
- Sends `SIGSEGV` signal to user process
- User process exits with “segmentation fault”
Summary

 Exceptions

- Events that require non-standard control flow
- Generated externally (interrupts) or internally (traps and faults)
- After an exception is handled, one of three things may happen:
  - Re-execute the current instruction
  - Resume execution with the next instruction
  - Abort the process that caused the exception
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

Diagram:

- Multiple memory sections labeled Stack, Heap, Data, Code
- Three CPU sections labeled Registers
- Indicates multiple processes running simultaneously
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

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

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

```
Process A                  Process B
| user code                | user code
|--------------------------|--------------------------
| save A                   | exception               | restore
| kernel code              | kernel code
| user code                | user code
| restore A                | save B                  | context switch
| kernel code              | kernel code
| user code                | user code
```

Assume only one CPU
Processes

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

Process 1

```
<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
</tr>
</tbody>
</table>
```

Process 2

```
<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
</tr>
</tbody>
</table>
```

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*”
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");
}
```

```
pid = Y
```

```
pid = 0
```

hello from parent  hello from child

*Which one appears first?*

non-deterministic!
Aside on the variable “pid”

- The variable name “pid” might be confusing (but is used in all examples here and in the book)
  - For the parent, it will hold the process id of the child
  - For the child, it will hold the value 0 (which is NOT the child’s process id)
  - Both processes have their own copy of the variable pid

- A more appropriate name for the variable might be: fork_return_val, which is what it actually holds.
  - 0 is returned to the child process
  - The process id of the child is returned to the parent process
  - A process can get its own process id by calling getpid()
Fork Example

```c
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) // child
        printf("Child has x = %d\n", ++x);
    else // parent
        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);
}
```

Possible
- C P C C
- BC BP P P
- P C BC BP
- BP BP BP BC

Not Possible
- C P
- BC BC
- BP C
- P BP

as long as C comes before BC and P comes before BP
**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");
        }
    }
    printf("Bye\n");
}
```

Seq 1: L0 L1 Bye Bye Bye L2

Seq 2: L0 L1 Bye L1 Bye L2 Bye Bye

A. No No
B. No Yes
C. Yes No
D. Yes Yes
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
// Example arguments: path="/usr/bin/ls",
void fork_exec(char *path, char *argv[]) {
    pid_t pid = fork();
    if (pid != 0) { // parent
        printf("Parent: created a child %d\n", pid);
    } else { // child
        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:

```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);
    }
}
```

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

- environ (global var)
  - envp (in %rdx)

- argv (in %rsi)
- argc (in %rdi)

Bottom of stack
Top of stack
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) { // child
        printf("HC: hello from child\n");
        exit(0);
    } else { // parent
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

feasible output: `HCHPCTBye`

infeasible output: `HPCTByeHC`
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 */
    }
}
```

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

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);
    }
}
```

```bash
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
    PID TTY       TIME CMD
6585 tty9    00:00:00 tcsh
6676 tty9    00:00:06 forks
6677 tty9    00:00:00 ps
linux> kill 6676
linux> ps
    PID TTY       TIME CMD
6585 tty9    00:00:00 tcsh
6678 tty9    00:00:00 ps
```

- 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
- \texttt{wait()} example
- \texttt{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
- L1
- Bye
- Bye

Infeasible output:
- L0
- Bye
- L1
- Bye
- L1
- 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

\texttt{pid\_t waitpid(pid\_t pid, int \&status, int options)}

- suspends current process until specific process terminates
- various options (that we won’t talk about)

```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 = 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);
    }
}
```