Assignments

- Reading for this section:
  - 8.1-8.4 (Processes); 8.5-8.6 (Signals; I/O); 10.1-10.4, 10.9 (Unix).
- Reading on the horizon: 4.2 (Digital logic).
- Be on the lookout for Project 4 (Multiprocessing)
- Quiz (Tue-Thu) on data structures (array addressing/struct alignment).
- Bonus Quiz (Fri-Sat) on processes.
- Midterm 2 (Mon, Nov 1st; **no class** that day).
  - Assembly (flow/calls/structures); processes/signals.
  - Similar conditions as the first exam (90min; open until 6pm).

Exceptional Behavior

Sometimes stack corruption may happen by accident, due to a bug.

What happens if we start executing pure noise?

What if something out of the ordinary happens?

- What if we execute an illegal instruction or access nonexistent memory?
- What if our computer overheats or just breaks?
- What if we press a key or get data over our WiFi?
- Or, what if we just want to set a breakpoint in our cdoe?

Standard Behavior of a Processor

What is it that makes exceptional behavior exceptional?

Normal Flow:

- A computer, fundamentally, executes one instruction at a time, in sequence.

Exceptional Flow:

- Reacting to an event that acts outside of – or goes beyond – normal flow.

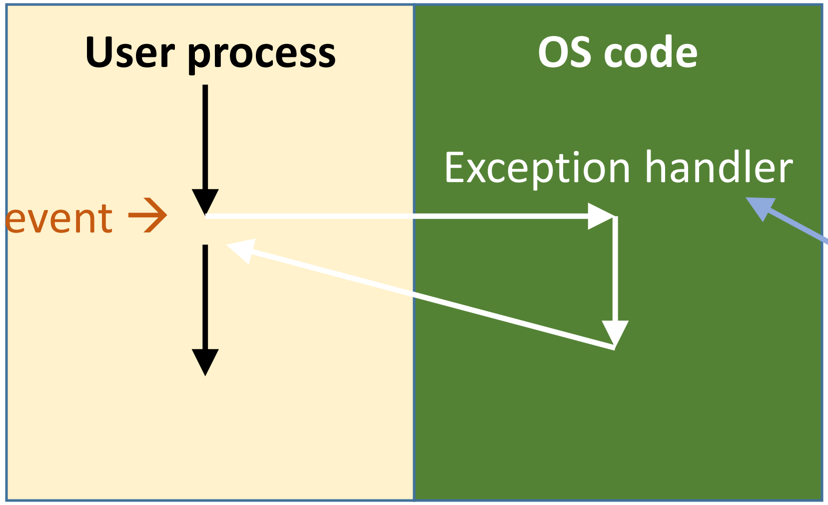
So how do we do it?

Exceptions

Exceptions will cause the program flow to divert.

When and exception is triggered, a predefined *exception handler* (or interrupt handler) is called.

Exceptional event →



Several different circumstances can trigger an exception.

An exception handler (somewhat) resembles a procedure call.

Differs from a procedure because of privilege level and other details.

Executing Exceptions

Exceptions are similar to a procedure call, but with more added on.

- We often can't predict an exception, so saving state is important.

Return address, flags, and stack address get saved automatically, so that the handler can leave system state intact when it's done.

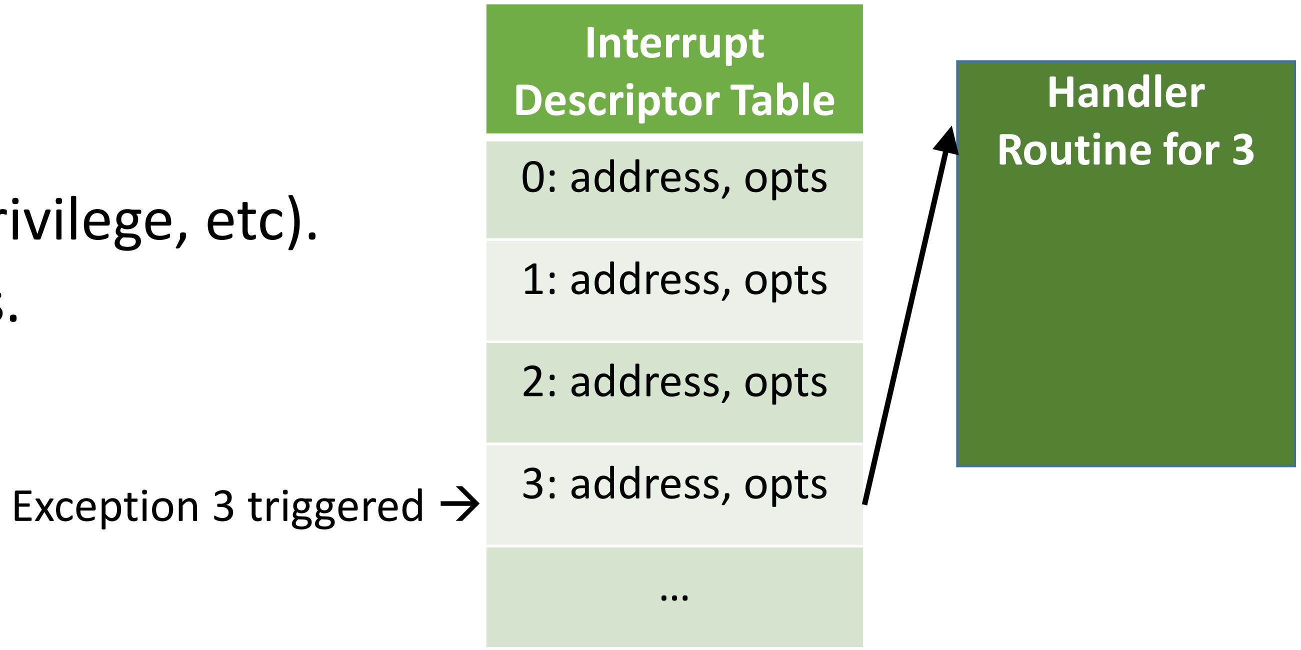
- We use a system table to look up the *exception handler*'s address by number.

Executing Exceptions, illustrated

Every exception type is numbered.

1. Interrupt k is triggered.
2. Look up entry k in the IDT.
   - Check options (switching stack, privilege, etc).
   - Get the interrupt handler address.
3. Call the handler.

Example: calling Interrupt 3.



Exception Table

When triggered, the handler address is found in the Exception Table.

- An Exception Table is a dedicated array in memory with all handler addresses.

In basic modes, the Exception Table is just a jump table (array of pointers).

- Called an *interrupt vector table* (IVT).

Modern machines can use a complex data structure for each exception type.

- Called an *interrupt descriptor table* (IDT).

Includes handler address, privilege level, which stack to use, additional options, etc.

Types of Exceptions: Traps

Calling `int $k` would call the handler for exception number k.

A *trap* is an intentionally triggered exception.

- Exceptions can be called in assembly using the `INT` instruction.

Example: Using a breakpoint while debugging a program.

- Done by temporarily replacing the code with `int $3`.

Example: Using a system call.

- Called using `int $80` (on Linux systems) or the `SYSCALL` instruction.

Like a procedure call, a trap proceeds to the next instruction after the call completes.

Types of Exceptions: Faults

Unintended, non critical exceptions are called faults.

- Generally caused as a side-effect of an illegal action.

Example: NULL pointer dereference, or dereference of non-existing memory.

```
movq $123, 0  # storing the value 123 at the address NULL
```

Example: Executing an undefined instruction.

Example: Divide-by-zero.

After the fault call, the system will attempt to re-execute the current instruction, or abort.

Types of Exceptions: Aborts

Severe, unrecoverable exceptions are called aborts.

- Generally happens when the computer is too broken to continue.

Example: Parity errors.
- Happens if memory was fundamentally corrupted (e.g. by cosmic rays).

Example: Machine check error due to an overheated computer.

An abort cause the current process to abort (cease entirely).

For severe errors, the safest thing to do is to stop everything as soon as possible, to limit damage.

Types of Exceptions: Interrupts

Exceptions due to asynchronous, outside events are called *interrupts*.

- Hardware *interrupt controller* may note an event and trigger an interrupt call.
- Software interrupts may be triggered by other processes.

Example: Pressing a key on a keyboard or moving the mouse.

Example: Incoming data on a network port.

Example: Ctrl-C is used on a process.

After an interrupt, the process resumes at the next line of code – unless explicitly prevented.

A process may be completely unaware that the interrupt even occurred!

Processes

What’s the difference between a program and a process?

Processes

What's the difference between a program and a process?

- A *program* is executable code stored on disk.
- A *process* is running executable code.

Process Abstractions

A process abstracts the concept of execution in two key ways:

Private address space.

- A program uses virtual memory to make it appear that it has all of memory to itself.
- Two different process may think they're accessing the same address, but it's not.

Logical Control Flow.

- A program behaves as if it is the only process currently running.
- May be accomplished by splitting time between multiple processes.

Processes Modes

Processes will typically be in one or two modes:

User Mode (or *non-privileged mode*).

- Only able to access user-mode instructions and process memory.

We cannot directly change the mode from an ordinary user process.

Kernel Mode (or *privileged mode*).

- Allows access to the full address space and full set of instructions.

The processor can enter Kernel Mode due to an interrupt or other exception.

Multiple Processes

A modern computer can run processes simultaneously.

- Processes whose flows overlap in time are called *concurrent processes*.
- Processes whose flows do not overlap are called *sequential processes*.

Multiple Processes

If a computer has multiple cores or multiple processors:

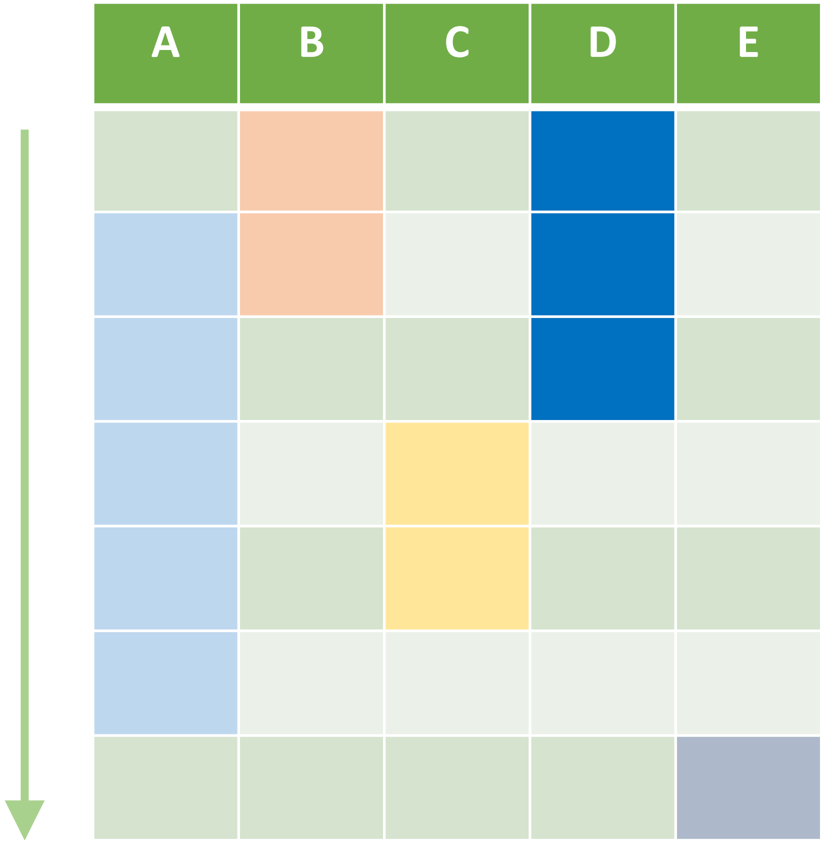
- Concurrent processing is easy.

If only one processes can use the CPU at one time:

- The processes must share.
- One process temporarily stops so that another one can resume.

Switching between processes on one processor is called *context switching*.

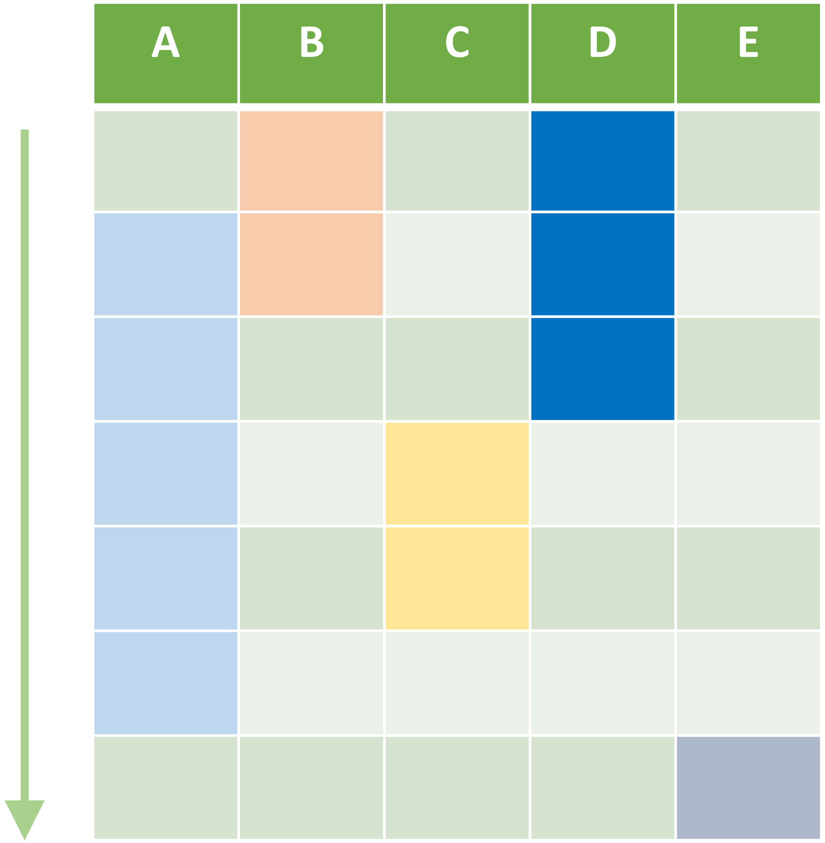
Multiple Processes, concurrency.



Which pairs of processes are concurrent?

Which pairs of processes are sequential?

Multiple Processes, concurrency.



Which pairs of processes are concurrent?

A-B; A-C; A-D; B-D

Which pairs of processes are sequential?

A-E; B-C; B-E; D-C; C-E; D-E

Forking Processes

We can spawn new processes in C using the `fork()` command.

- A *fork* creates an exact duplicate of a process and its memory space.
- The original version (*parent*) and the forked copy (*child*) run concurrently.

The copy is a full copy, but independent of the original.

The copy action may be deferred (for performance) if it data is used in a read-only way.

Synopsis:

```
#include <unistd.h>
int fork(void);
```

Forking Processes, return value

The `fork()` function returns a value for both the parent and the child.

- The return value is how we discover whether we are the parent or the child!

The parent learns the Process ID (PID) of the child process.

The child receives the value 0.

Example:

```
int main() {
        if (fork()) printf("We are the parent.\n");
        else printf("We are the child.\n");
        return 0;
}
```

Forking Processes, example

Parent process:

Child PID →

```c
int main() {
  if (fork())
    printf("We are the parent.\n");
  else
    printf("We are the child.\n");
  return 0;
}
```

Child process:

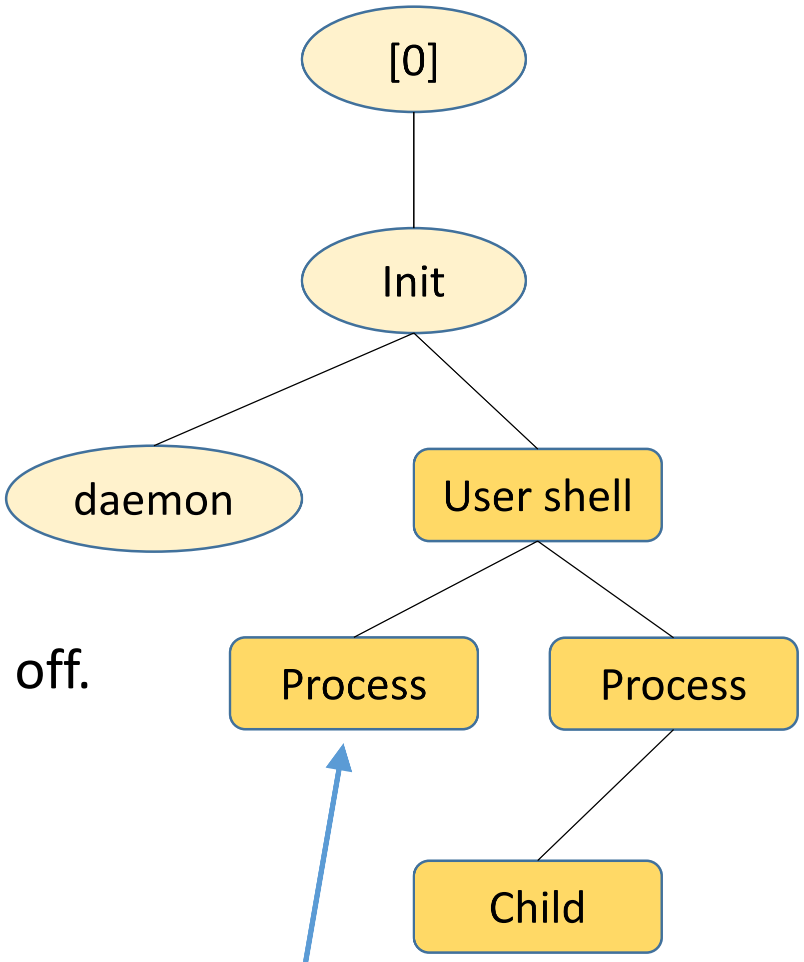
```c
int main() {
  if (fork())
    printf("We are the parent.\n");
  else
    printf("We are the child.\n");
  return 0;
}
```

← *Zero*

A fork() call is made once, but effectively returns twice!

Notes about Forks

A `fork()` is how we create new processes.



After the fork, both processes run independently.

- The process starts in the same place where the parent left off.

There are no guarantees about which one runs more quickly.

A forked child is a seemingly exact duplicate of the parent.

It uses a deep copy of parent memory.

- The copy may be deferred to the first write.

Our user processes are children of other processes (e.g. the shell and the init system).

Process Lifespan

1. New process is created using `fork()`.

   The child process runs independently of the parent, with no guarantees about the order of execution.

2. The process transforms into a new program using `exec()`.

   The exec family of calls have several variangs, e.g. `execl()`.

3. The process terminates with `exit()` or a return from `main()`.

   We can use the `atexit()` function to call a handler on exit.

4. The parent can reap the child using `wait()` or `waitpid()`.

   This step is optional:
   It would happen anyways when the parent terminates.

Reaping a process lets us find out it’s exit return code.

Process Tracking

Assume this code.

```
int main() {
	printf("pre\n");
	if (fork()) {
		printf("mid\n");
		fork();
	}
	printf("end\n");
	return 0;
}
```

What gets executed by whom?

- Multiple outputs are possible.
- Only some outputs are legal.

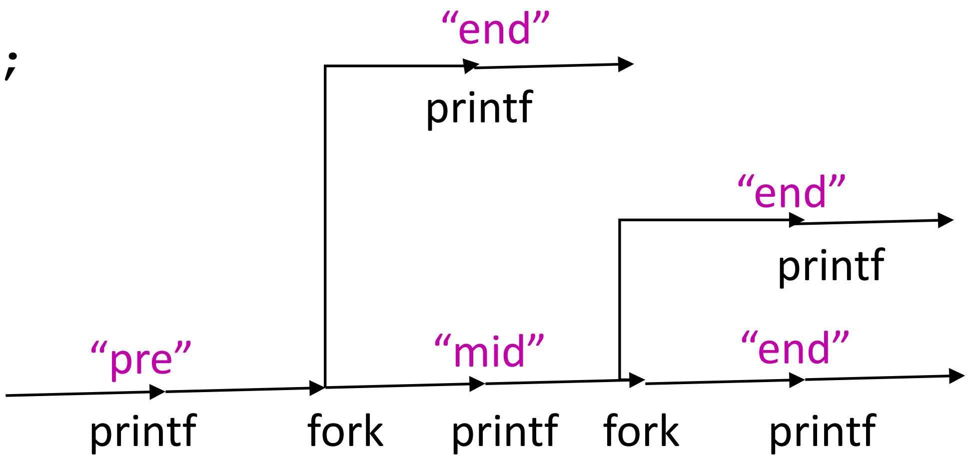
Process Tracking

Assume this code.

```
int main() {
    printf("pre\n");
    if (fork()) {
        printf("mid\n");
        fork();
    }
    printf("end\n");
    return 0;
}
```

What gets executed by whom?

- Multiple outputs are possible.
- Only some outputs are legal.
- Helps to use a *process graph*.



Once a process forks, there's no guarantee which branch runs faster.
What orders of execution might be legal?

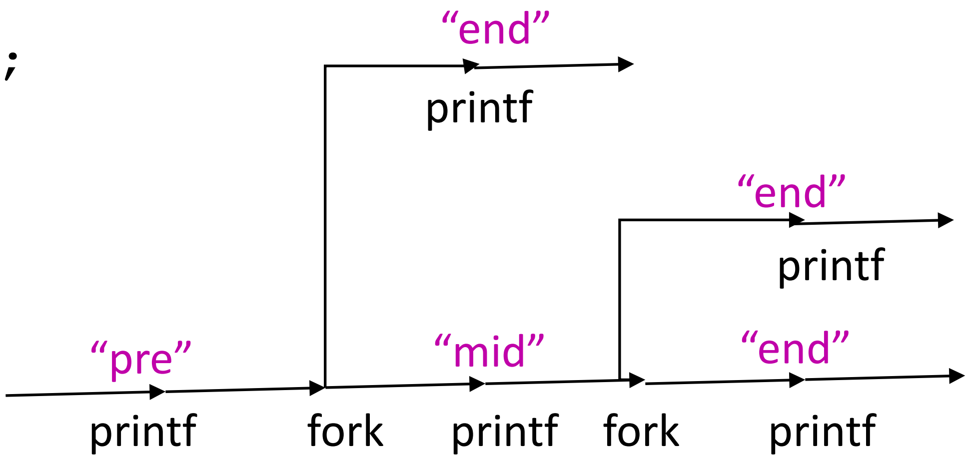
Process Tracking

Assume this code.

```
int main() {
        printf("pre\n");
        if (fork()) {
                printf("mid\n");
                fork();
        }
        printf("end\n");
        return 0;
}
```

What gets executed by whom?

- Multiple outputs are possible.
- Only some outputs are legal.
- Helps to use a *process graph*.



Legal output: `pre, mid, end, end, end`

Legal output: `pre, end, mid, end, end`

Illegal output: `pre, end, end, mid, end`

There's no way for a second "end" to appear before the "mid".

Running New Programs

New programs are started using an *exec* command, after the process is created.

We'll often see a `fork()` immediately followed by an exec call.

If successful, the exec *never returns*.

- Instead, it overwrites the current process with the new program.

Should match The program name

Additional command line args start here

Synopsis:

```
#include &lt;unistd.h&gt;
int execl(char* prog, char* arg1, char* arg2,…);
```

Example:

```
execl("/usr/bin/ls", "ls", "-lR", "cs367/", NULL);
```

NULL terminate

Full path

Ending a Program

We can terminate a program at any time using `exit()`.

Calling `atexit()` prior to exiting will register a cleanup function.

Ending `main()` with an integer return value has the same effect.

Synopsis:

```
#include <stdlib.h>
void exit(int status);
```

We provide an integer exit code.

- A code of 0 means normal termination.
- Anything else indicates failure.

Waiting for Processes to Finish

If we want to wait for a child process to finish, we can use `wait()`.

- It will cause the program to wait until some child is finished.
- We can use this to synchronize two processes.

The `wait` instruction waits for the first available process to finish.

Synopsis:

```
    #include <sys/wait.h>
pid_t wait(int *stats);
pid_t waitpid(pid_t pid, int *stats, int options);
```

Status of the reaped process, or NULL if we don't care.

Child PID; or -1 on error

0 if WNOHANG and no ready child

PID to wait on, or -1 if we don't care.

0 for normal `wait()` behavior; `WNOHANG` to return immediately if no child is ready.

The `waitpid` waits for a specified process to finish.

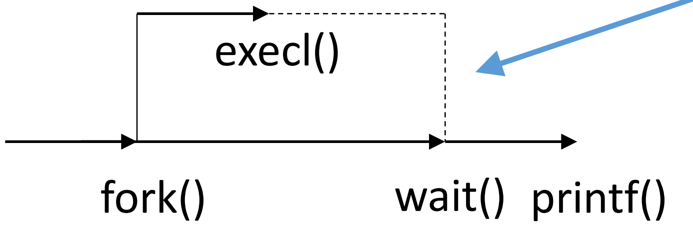
Waiting: Synchronizing

Here, we use `wait()` to ensure that a child process completes.

Example: Run a different command as the child, then print a message afterwards as parent.

```
int main() {
    if (fork()) {
            wait(NULL);     // we are the parent, wait for the child to finish
            printf("That's all, folks!\n");
    } else {
            execl("/usr/bin/ls", "ls", ".", NULL);       // we are the child
    }
    return 0;
}
```

A dotted line in the process graph indicates a `wait()`.



Zombie Processes

When a program ends or exits, it doesn’t automatically disappear.

- First, it becomes a *zombie process*.

A zombie process is no longer running, but still has a presence.

- A zombie can be used by a parent to extract exit status information.

A defunct process has been *reaped* if its status is read and the process is removed.

Waiting: Reaping Exit Status

Here, we use `wait()` to find out how the child exited.

```
int main() {
        int p, status;
        for (p = 0; p < 10; p++) { if (!fork()) exit(p + 50); }      // only children will exit
        for (p = 0; p < 10; p++) {
                pid_t pid = wait(&status);          // wait for a child to finish
                if (WIFEXITED(status))              // if normal exit
                        printf("child %d terminated, status = %d\n", pid, WEXITSTATUS(status));
                else printf ("child %d terminated abnormally\n", pid);
        }
        return 0;
}
```

Macro which tells us whether we exited normally or not

Macro which extracts the exit code from the status value

Shells

Command shells allow us to type commands and execute them.

- The shell spawns (forks) a new job (process), then uses exec.
- The job may run in the *foreground* or *background*.

Waits for the job to finish, using `waitpid()`.

Lets the child process run without waiting for it.

Communicating with Processes: Signals

One very simple way to communicate with processes is with *signals*.

Examples: Ctrl-C (`SIGINT`); timer alarm (`SIGALRM`); kill process (`SIGKILL`).

Signals are simplistic: there is no associated message or data.

A process only learns that *a signal has been sent* and *which type of signal* it is.

Signals **do not queue** if multiple signals of the same type are sent.

A process only discovers *whether* it has been signaled, not *how many times*.

Signals are closely related to the exception mechanism.

Signals can be sent by one process to another, but they are administered by the kernel.
Often they occur as a response to exceptions.

Signals, examples

Signal name	Signal value	Purpose	Default action
`SIGINT`	2	Ctrl-C from keyboard	Terminate the process
`SIGKILL`	9	Kill process	*Terminate the process**
`SIGSEGV`	11	Invalid memory reference (segfault)	Terminate & dump core
`SIGALRM`	14	Timer signal	Terminate the process
`SIGTERM`	15	Polite termination request	Terminate the process
`SIGCHLD`	17	Child stopped or terminated	Ignore
`SIGSTOP`	19	Stop (suspend) process	*Stop the process**
`SIGTSTP`	20	Ctrl-Z from keyboard	Stop the process

`SIGSTOP` and `SIGKILL` are special: their default action **cannot be altered**.
They cannot be caught, blocked or ignored.

Signals, sending

There are several ways to send signals to a process:

1. Indirectly as a result of another action.

   Example: Pressing Ctrl-C (generates `SIGINT`).

   Example: Using the `alarm()` function (generates `SIGALRM`).

2. From the command shell using the `kill` command.

   Example: `kill 123                # sends process 123 a SIGTERM signal`

   Example: `kill -9 456             # sends process 456 a SIGKILL signal`

3. From C using the `kill()` function.

Yes, it is horribly named, because it is used for any signal type.

Synopsis:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig); //ex: kill(456, 9) will send process 456 a SIGKILL
```

Handling Signals

Signals can be *caught* and *handled* in some cases.

- Default handlers (software routines) are provided for all signals.

- Many signal handlers can be overridden by custom routines.

To do this, we must *install* a new signal handler.

- Signals can also be ignored or (temporarily) blocked.

Exceptions: SIGKILL and SIGSTOP *cannot be caught, blocked, or ignored*.

Installing Signal Handlers

To install a new signal handler routine, we must first create the handler.

Declare a handler routine with an `int` input and `void` output.

Use `sigaction()` to install the new handler.

Synopsis:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction* act, struct sigaction*oldact);
```

Now the registered handler will be called whenever the signal occurs.

- *Note that repeated calls might only be triggered once.*

Signal type to listen for.

The structure which includes the address of the handler.

Installing Signal Handlers, example

This may only get called once for multiple children (no queue).

To mitigate the problem, we can use a loop, then waitpid() until there are no more.

Creating a handler to listen for finishing children:

```
static int numprocs = 5;
void child_handler(int sig) {
     wait(NULL);                        // wait to reap the process we were just alerted to
     numprocs--;                        // update the running counter condition
}
int main() {
     struct sigaction act;              // for storing signal handler overhead
     memset(&act, 0, sizeof(act));      // initialize to zero
     act.sa_handler = child_handler;    // child_handler will be our signal handler
     sigaction(SIGCHLD, &act, NULL);    // sets a new interrupt handler

     for (int j = 0; j < numprocs; j++)
     { if (!fork()) exit(0); }          // fork several dummy processes

     while (numprocs) {  }              // "infinite" loop
     return 0;
}
```

We can set this to SIG_IGN if we want to ignore the signal, or SIG_DFL if we want to restore the default.

Signal-Related Functions

```
// blocks process for the given # of seconds
unsigned int sleep(unsigned int seconds);

// suspends the process
int pause(void);

// gets the current process's ID
pid_t getpid(void);

// sends a signal to the specified process
int kill(pid_t pid, int sig);

// equivalent to kill(getpid(), sig)
int raise(int sig);
```

Process Communication with Pipes

Signals are a simple, but not very informative way to communicate.

What if a process wants to share detailed information with another one?

A pair of processes can send information over a *pipe*.

A pipe is like a 2-ended file: write into one end, read from the other.

The ends of the pipe behave like an ordinary files (`read`, `write`, etc).

Bidirectional communication requires two pipes.

Unix Process Abstractions

Pipes work because Unix makes many things resemble files.

Examples of things which can be abstracted as file descriptors:

- Pipes.
- Ordinary files.
- Input/output streams.
- Sockets.

Keyboard input or terminal output.

A file is a sequence of bytes.

- Files can be ASCII/Unicode text.
- Files can be raw binary.

Sockets are used for network connections.

Standard File Descriptors

Three file descriptors are created when we run any process:

Category	Data stream	File descriptor	Descriptor named	Default location
Standard input	`stdin`	0	`STDIN_FILENO`	Keyboard
Standard output	`stdout`	1	`STDOUT_FILENO`	Terminal window
Standard error	`stderr`	2	`STDERR_FILENO`	Terminal window

- Yes, it is possible to change these from within a process.
- A child inherits the descriptors their parents are currently using.

Descriptors vs Streams

Notice that we can refer to files using both *file descriptors* and *streams*.

File descriptors are numbers - e.g. `STDOUT_FILENO`, which is FD 1.

FDs use commands like: `open()`, `read()`, `write()`, `close()`.

File descriptors are a Unix-centric concept.

Streams are type `FILE*`, e.g. `stdout`.

Streams use commands like: `fopen()`, `fgets()`, `fprintf()`, `fclose()`.

Streams are a standard C concept.

Streams are buffered wrappers around raw file descriptors.

- We can create new streams from FDs using `fdopen()`.
- `fdopen()` is invoked like `fopen()`, with a FD instead of a file name.

Using File Descriptors

Opening, reading, writing, and closing files:

```
#include <sys/types.h>    // for open
#include <sys/stat.h>     // for open
#include <fcntl.h>        // for open
#include <unistd.h>       // for read, write, and close

int open(const char *pathname, int flags);
        // ex: int input = open("input.txt", O_RDONLY);
ssize_t read(int fd, void* buf, size_t count);
        // ex: int total_read = read(input, buf, sizeof(buf));
ssize_t write(int fd, const void* buf, size_t count);
        // ex: int total_write = write(output, buf, total_read);
int close(int fd);
        // ex: close(input);
```

`O_RDONLY` – open for reading
`O_WRONLY` – open for writing
`O_RDWR` – open read/write
`O_TRUNC` – clear file if exists
`O_APPEND` – append to existing
`O_CREAT` – create if not exists

File to open

File to open

Maximum bytes to read/write

Bytes acturally read/written

Place to store data read / data to write

Notes about FDs and Signal Safety

Signals are triggered asynchronously.

A process's signal handler can be called at any time.

- It can potentially disrupt an instruction before it is finished.

Example: Our handler gets called during an I/O call:

- The error code had been set during the call.
- The handler may change the error code. We've lost our real error code!

We must always think about making our handlers safe.

Using `read`/`write` is safer than `fgets`/`printf`, due to side effects.

Notes about Signal Safety

It may be a good idea to *block* a signal while working with key data.

- A blocked signal will be marked but not delivered until it is unblocked.

- Use `sigprocmask()` to temporarily block a signal:

```
sigset_t mask, prev;
sigemptyset(&mask)                        // init the mask to empty
sigaddset(&mask, SIGINT);                 // create a mask for SIGINT
sigprocmask(SIG_BLOCK, &mask, &prev);     // block SIGINT;
                                          // save old mask for later
// code while blocked goes here
sigprocmask(SIG_SETMASK, &prev, NULL);    // restore the old mask
```

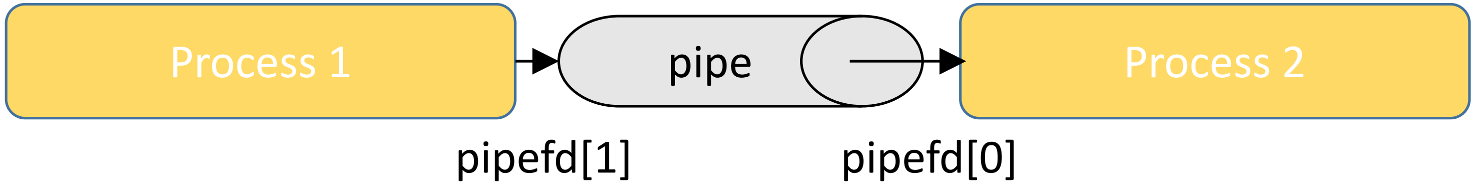
Blocking a signal will prevent it from interrupting flow mid-update.

Creating Pipes

To create a pipe (a read/write pair), we use the `pipe()` command.

- A child inherits both ends of a pipe, and can close the end it doesn't need.

Usage:
1. Create the pipe.
2. `fork()` a child, which inherits the pipe.



Synopsis:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

After the call, `pipefd[1]` is for writes to the pipe, and `pipefd[0]` is for reads from the pipe.
Tip: This is the same numbering as `stdin/stdout`.

The call will fill this array with two new file descriptors.

Changing Existing Streams

In some cases, we may want to change an existing stream.

- We replace one file descriptor with another file descriptor.
- To do this, we use the `dup2()` command.

Synopsis:

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

If the FD being replaced is already open, it will close it first.

It makes sense to close the old FD after transferring to the new one.

Example: We want a child to use a different `stdin`.
We open “`file.txt`” for reading, using `fd` as a descriptor.
We’d use `dup2(fd, STDIN_FILENO)` to read from the file instead of the terminal.

Shell Redirection Example, input

Example of a shell command: `sort < items.txt`

```
int main() {
        if (!fork()) {  // only do this for the child:
                // open the input file as a read-only file
                int file = open("items.txt", O_RDONLY);
                // transfer the new file descriptor into stdin
                dup2(file, STDIN_FILENO);
                // now that this process reads from the file to get its
                // standard input, transform this into a sort process
                execl("/usr/bin/sort", "sort", NULL);
        }

        return 0; // never reached by the child
}
```

Shell Redirection Example, output

Example of a shell command: `ls > contents.txt`

```
int main() {
        if (!fork()) {  // only do this for the child:
                // open the output file as a write-only file
                int file = open("contents.txt", O_WRONLY|O_TRUNC|O_CREAT);
                // transfer the new file descriptor into stdout
                dup2(file, STDOUT_FILENO);
                // now that this process writes to the file when it produces
                // standard output, transform this into an ls process
                execl("/usr/bin/ls", "ls", NULL);
        }

        return 0; // never reached by the child
}
```

Shell Redirection Example, output

Example of a shell command: `ls >> contents.txt`

Append instead of starting over

```
int main() {
        if (!fork()) {  // only do this for the child:
                // open the output file as a write-only file
                int file = open("contents.txt", O_WRONLY|O_APPEND|O_CREAT);
                // transfer the new file descriptor into stdout
                dup2(file, STDOUT_FILENO);
                // now that this process writes to the file when it produces
                // standard output, transform this into an ls process
                execl("/usr/bin/ls", "ls", NULL);
        }

        return 0; // never reached by the child
}
```

Shell Pipe, example

Example of a shell command: `ls | sort -r`

```
int main() {
        int pipefd[2];      // will store
        pipe(pipefd);       // create the two ends of a new pipe
        if (!fork()) {      // execute only for the child:
                dup2(pipefd[1], STDOUT_FILENO);        // one end of the pipe will be our stdout
                close(pipefd[0]);                      // unneeded by this child
                execl("/usr/bin/ls", "ls", NULL);      // run "ls" with the output being piped
        }
        if (!fork()) {      // execute only for the child:
                dup2(pipefd[0], STDIN_FILENO);         // one end of the pipe will be our stdin
                close(pipefd[1]);                      // unneeded by this child
                execl("/usr/bin/sort", "sort", "-r", NULL);      // run "sort" with the input being piped
        }
        return 0;           // only the parent reaches this spot
}
```