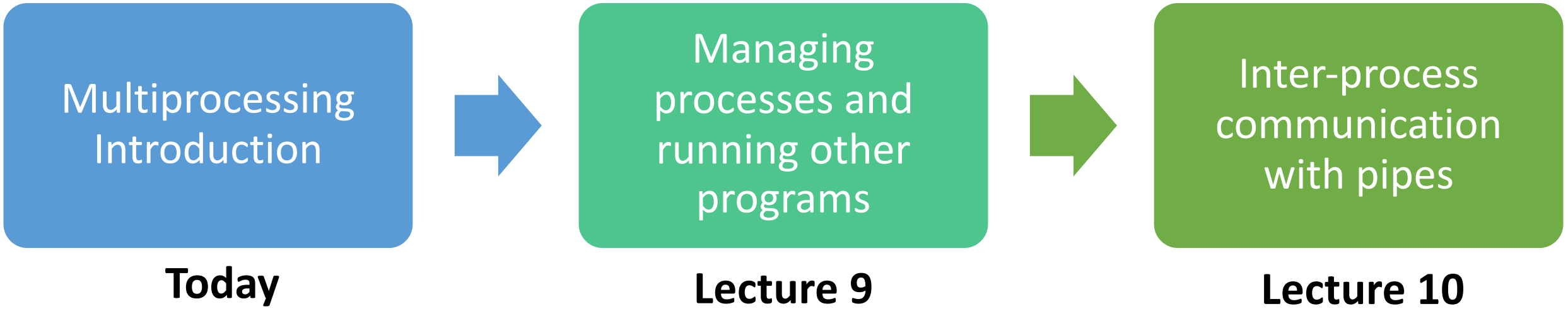
CS111, Lecture 8
Multiprocessing Introduction

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 4

Topic 2: Multiprocessing - How can our program create and interact with other programs? How does the operating system manage user programs?

CS111 Topic 2: Multiprocessing



assign3: implement your own shell!

Learning Goals

- Understand the limitations and tradeoffs of crash recovery
- Learn how to use the **fork()** function to create a new process
- Understand how a process is cloned and run by the OS

Plan For Today

- **Finishing up**: Crash Recovery
- Multiprocessing overview
- Introducing **fork()**

Plan For Today

- **Finishing up: Crash Recovery**
- Multiprocessing overview
- Introducing **fork()**

Crash Recovery

We've discussed 3 main approaches to crash recovery:

1. **Consistency check on reboot (fsck)** – no filesystem changes, run program on boot to repair whatever we can. But can't restore everything and may take a while.
2. **Ordered Writes** – modify the write operations to always happen in particular orders, eliminating various kinds of inconsistencies. But requires doing synchronous writes or tracking dependencies and can leak resources.
3. **Write-Ahead Logging** – log metadata (and optionally file data) operations before doing the operations to create a paper trail we can redo in case of a crash.

Write-Ahead Logging ("Journaling")

Problem: log can get long!

Solution: occasional "checkpoints" – truncate the log occasionally once we confirm that portion of the log is no longer needed.

Problem: could be multiple log entries for a single "operation" that should happen atomically.

Solution: have a log mechanism to track "transactions" (atomic operations) and only replay those if the entire transaction is fully entered into the log. (assign2 wraps each transaction with LogBegin and LogCommit)

Problem: we could replay a log operation that has already happened.

Solution: make all log entries *idempotent* (doing multiple times has same effect as doing once). E.g. "append block X to file" (bad) vs. "set block number X to Y"

Write-Ahead Logging ("Journaling")

Problem: log entries must be written synchronously before the operations

Solution: delay writes for log, too (i.e. build log, but don't write immediately; when a block cache block is written, write relevant log entries then). Though this risks losing some log entries.

Logging doesn't guarantee that everything is preserved, but it does guarantee that what's there is consistent (separates *durability* – data will be preserved – from *consistency* – state is consistent)

Crash Recovery

Ultimately, tradeoffs between *durability, consistency* and *performance*

- E.g. if you want durability, you're going to have to sacrifice performance
- E.g. if you want highest performance, you're going to have to give up some crash recovery capability
- What kinds of failures are most important to recover from, and how much are you willing to trade off other benefits (e.g. performance)?

Still lingering problems – e.g. disks themselves can fail

Demo – Filesystem Recovery

- Assign2 tools let you simulate real filesystems, make them crash, and experiment with recovery tools
- Implement a program that replays a log after a crash
- Mix of filesystem exploration (playing around with simulated filesystems, viewing logs and filesystem state) and coding (about ~10-15 lines total)
- You'll have a chance to play with these tools in the assignment and in section this week. Let's see a demo!

Demo – Filesystem Recovery

In assign2 you can create, interact with, corrupt and recover Unix v6 filesystems. Demo (in assign2 starter project):

1. **`./mkfsv6 v6.img`** (makes a new Unix v6 filesystem image called v6.img)
2. **`mkdir mnt`** (makes a folder **mnt** where we will "mount" the filesystem)
3. **`CRASH_AT=100 ./mountv6 -j v6.img mnt &`** (makes the **v6.img** filesystem image appear in the folder **mnt.** "&" runs in the background. "-j" adds journaling. CRASH_AT=100 crashes it after 100 block-write operations.
4. **`cd mnt`** (go into **mnt** to explore the filesystem image)
5. **`touch `seq 1000``** (makes 1000 empty files, named 1 to 1000)
6. **`cd ..`** (exit crashed filesystem directory)
7. **`cp v6.img v6-2.img`** (make a copy of v6.img called v6-2.img to test recovery mechanisms)

Demo – Filesystem Recovery

In assign2 you can create, interact with, corrupt and recover Unix v6 filesystems. Demo (in assign2 starter project):

8. **`./fsckv6 -y v6.img`** (run fsck on v6.img and repair it)
9. **`./samples/apply_soln v6-2.img`** (run log recovery on v6-2.img and repair it)
10. **`./mountv6 -j v6.img mnt &`** (mount v6.img again to see fsck results)
11. **`cd mnt`** (examine filesystem to see which of the 1-1000 files are there)
12. **`cd ..`** (exit filesystem when done)
13. **`fusermount -u mnt`** (unmount v6.img filesystem)
14. Repeat steps 10-13 for **v6-2.img** to compare logging recovery vs. fsck
15. **`./dumplog v6.img`** (view filesystem log for **v6.img**)

Plan For Today

- **Finishing up**: Crash Recovery
- **Multiprocessing overview**
- Introducing **fork()**

Multiprocessing Terminology

Program: code you write to execute tasks

Process: an instance of your program running; consists of program and execution state.

Key idea: multiple processes can run the same program

Process 5621

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    printf("Goodbye!\n");
    return 0;
}
```

Multiprocessing

Your computer runs many processes simultaneously - even with just 1 processor core (how?)

- "simultaneously" = switch between them so fast humans don't notice
- Your program thinks it's the only thing running
- OS *schedules* tasks - who gets to run when
- Each gets a little time, then has to wait
- Many times, waiting is good! E.g. waiting for key press, waiting for disk
- *Caveat*: multicore computers can truly multitask

Playing with Processes

When you run a program from the terminal, it runs in a new process.

- The OS gives each process a unique "process ID" number (PID)
- PIDs are useful once we start managing multiple processes
- **getpid()** returns the PID of the current process

```c
// getpid.c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
      pid_t myPid = getpid();
      printf("My process ID is %d\n", myPid);
      return 0;
}
```

```
$ ./getpid
My process ID is 18814

$ ./getpid
My process ID is 18831
```

Plan For Today

- **Finishing up**: Crash Recovery
- Multiprocessing overview
- **Introducing fork()**

fork()

fork() creates a second process that is a **clone** of the first:

```
pid_t fork();
```

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

Process A

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

```
$ ./myprogram
Hello, world!
Goodbye!
Goodbye!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
```

fork()

fork() creates a second process that is a **clone** of the first:

```
pid_t fork();
```

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
Goodbye, 2!
Goodbye, 2!
```

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

- **parent** (original) process forks off a **child** (new) process
- The child **starts** execution on the next program instruction. The parent **continues** execution with the next program instruction. *The order from now on is up to the OS!*
- **fork()** is called once, but returns twice (why?)

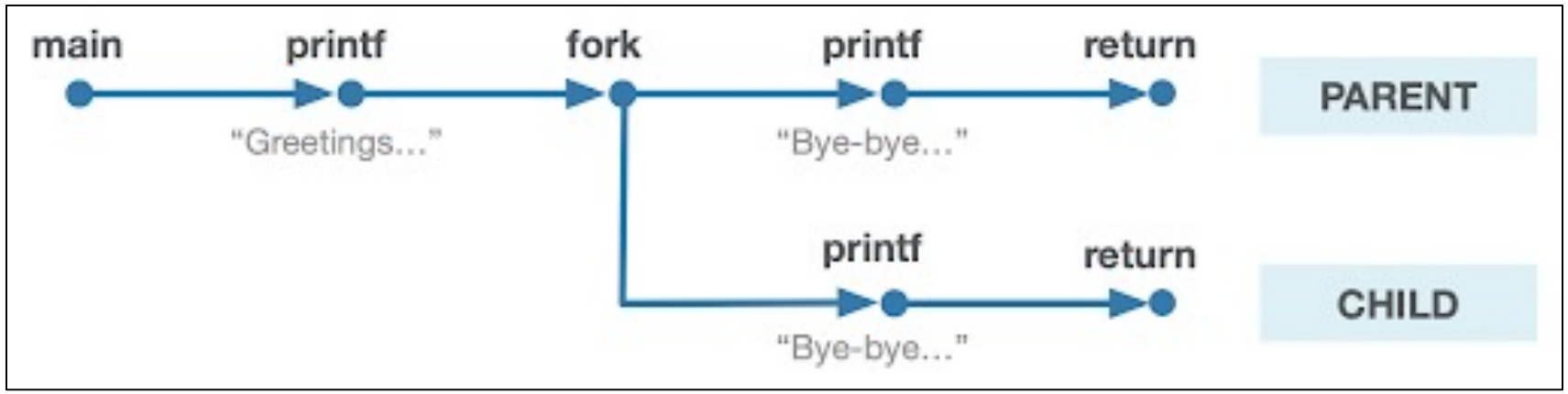


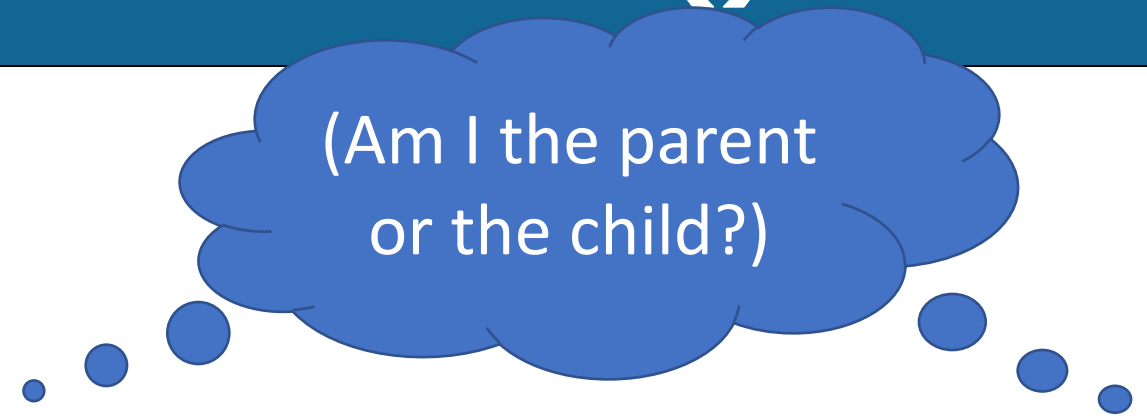
Illustration courtesy of Roz Cyrus.

fork()

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

- **parent** (original) process forks off a **child** (new) process
- Everything is duplicated in the child process (except PIDs are different)
  - File descriptor table - this explains how the child can still output to the same terminal!
  - Mapped memory regions (the address space) - regions like stack, heap, etc. are copied

fork()



Process A

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Process B

```
int main(int argc, char *argv[]) {
    int x = 2;
    printf("Hello, world!\n");
    fork();
    printf("Goodbye, %d!\n", x);
    return 0;
}
```

Is there a way for the processes to tell which is the parent and which is the child?

fork()

Key Idea: **the return value of fork() is different in the parent and the child.**

fork() creates a second process that is a **clone** of the first: `pid_t fork();`

- **parent** (original) process forks off a **child** (new) process
- In the **parent**, **fork()** will return the PID of the child (only way for parent to get child's PID)
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

Process 112

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

Process 112

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
fork returned 112
fork returned 0
```

fork()

In the **parent**, **fork()** will return the PID of the child. In the **child, fork()** will return 0 (this is not the child's PID, it's just 0).

Process 111

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

Process 112

```
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    pid_t pidOrZero = fork();
    printf("fork returned %d\n",
            pidOrZero);
    return 0;
}
```

```
$ ./myprogram
Hello, world!
fork returned 112
fork returned 0
```

OR

```
$ ./myprogram
Hello, world!
fork returned 0
fork returned 112
```

We can no longer assume the order in which our program will execute! The OS decides the order.

fork()

- In the **parent**, **fork()** will return the PID of the child
- In the **child, fork()** will return 0 (this is not the child's PID, it's just 0)
- if **fork()** returns < 0, that means an error occurred
- **getppid()** gets the PID of your parent and **getpid()** gets your own PID
- This is how your shell works – shell (parent) forks off child process to run a command you enter. When you run a command, its parent is the shell.

Recap

- **Finishing up**: Crash Recovery
- Multiprocessing overview
- Introducing **fork()**

Next time: more about fork, plus how to wait for child processes to finish, and how to run other programs.

Lecture 8 takeaway: fork() allows a process to fork off a cloned child process. The order of execution between parent and child is up to the OS! We can distinguish between parent and child using fork's return value (child PID in parent, 0 in child).