CSci 4061
Introduction to Operating Systems

Processes in C/Unix

Chapter 3 (R&R)

Process as Abstraction

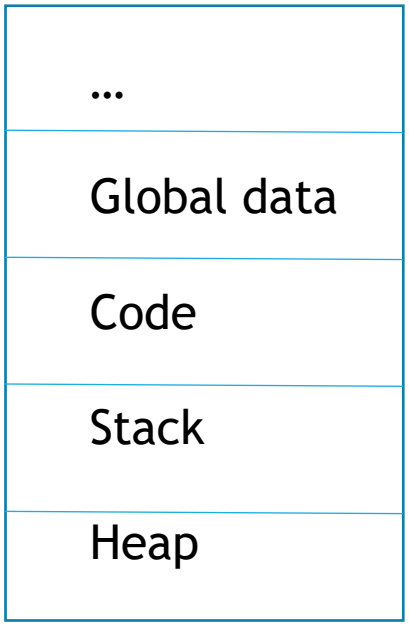
- Talked about C programs a bit
- Program is a static entity
- Process is an abstraction of a running program provided by the OS
  - Granted resources

Process Abstraction

- Process operations
  - Create: `fork`
  - Change: `exec*`
  - Terminate: `exit/abort/return`/signals
  - Synchronize: `wait*`, and others (later)

Virtual Address Space

- Memory map of the process
- Virtual address space (VAS)
  - Set of legal addresses

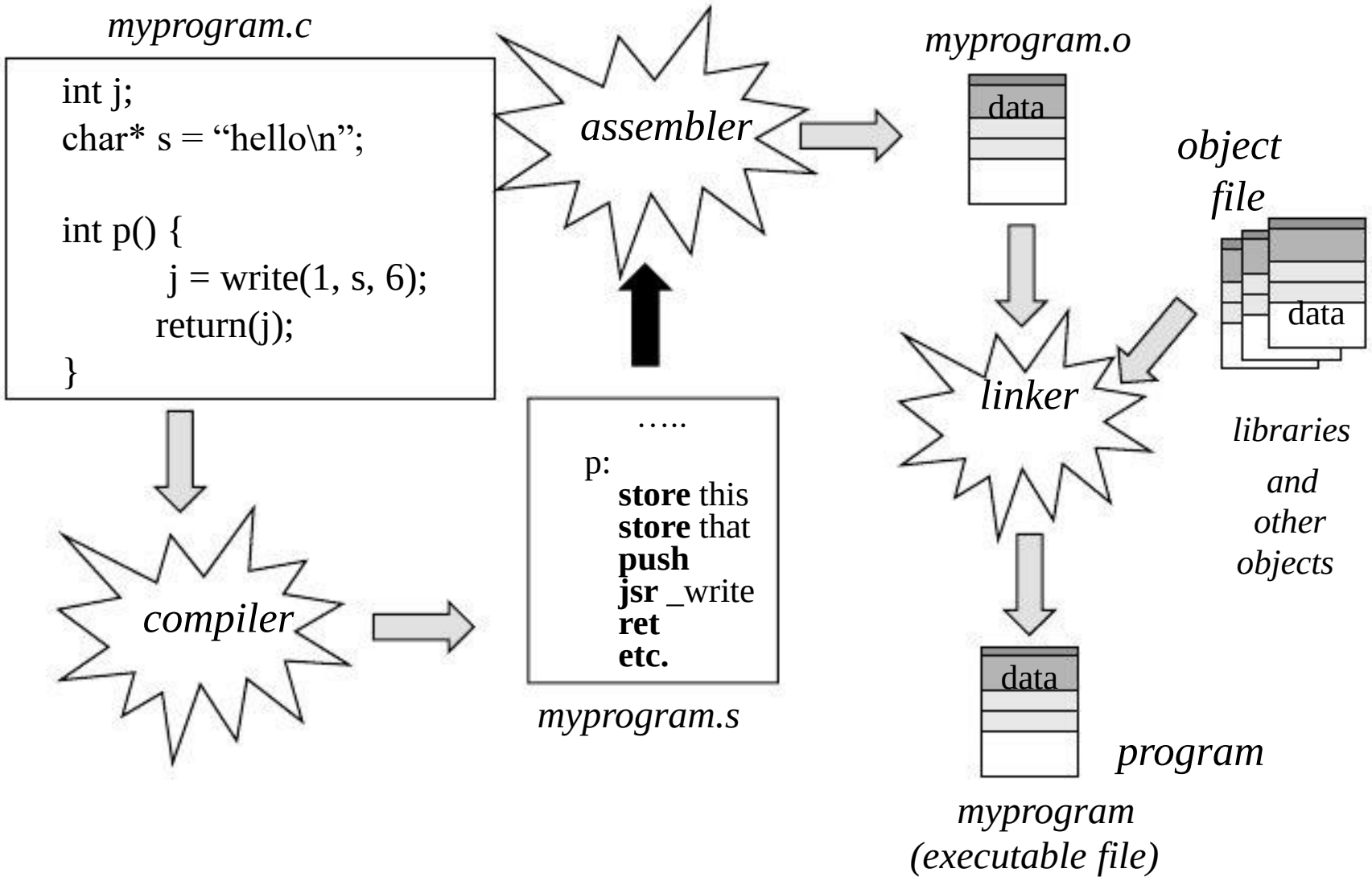


Address space disjoint

Isolation

Top-down: Why Processes?

The Birth of a Process: Executable



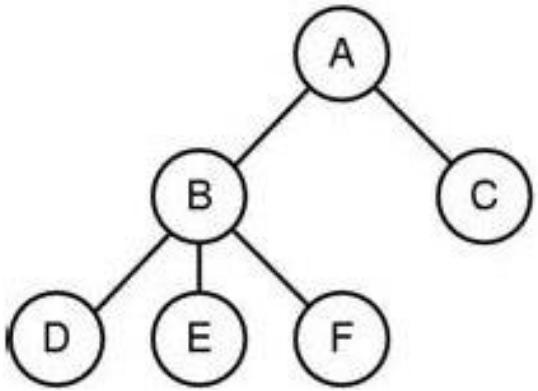
Unix Processes

- Process trees
- Creation: `fork/exec`
- Synchronization/Control: `wait`
- Termination: `exit`
- Error handling
- Identities

Processes in Unix

- Processes in Unix form a hierarchy
  - Root is called *systemd*, started by OS
  - Child is created by parent
  - Relationship important for communication

- Unix shell is a child of *systemd*, shell in turn creates processes



“Your process”

- When you log in …
- A process known as your shell is created and running
- Has an associated terminal window for I/O
- It is a foreground process and allows you to interact with it

Processes in Unix (cont'd)

shell> `cat file1 file2`

… output

shell>

Example of a **foreground** process: shell waits for completion + user can interact with process

Under the hood:

- shell creates "cat" process
- waits for it to finish
- prompts for next command

Processes in Unix (cont'd)

shell>`my_program`**&** // & put in the background

shell>`cat foo.c`

Background process: shell does not wait for `my_program` to complete before next prompt; user cannot interact with background process

Examples?

Switching:foreground to background

- `<loop>`
- ^z to suspend a process; get control back
- `bg` run it in the background
- `fg` run it in the foreground

- process shell creates -- *job*
- `jobs`: lists job ids (different from process ids) in current shell only (`top`, `ps` machine-wide)

System Programming Interface: `fork`

- The Unix system call for process creation is called `fork()`

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(); // pid_t is a process id (#)
```

- With any Unix system call: error codes, header files, parameters

- The `fork` system call creates a “clone” of the parent process

Fork (cont'd)

- Child is given a "copy" of the parent's memory (virtual address space)
  - actually a virtual copy using copy-on-write
- Child is running the same program code as the parent
- Child begins life with the same register values of the parent (e.g. PC)
- Child inherits resources from the parent
  - open files
- What is not shared?
  - locks, IDs, signals, CPU time measures, …

PARENT

```
#include <unistd.h>
int pid;
int status = 0;
pid = fork ();
// Parent: PID IS NON-ZERO
if (pid > 0) {
 printf ("Parent: child has
          pid=%d", pid);
         …..
 pid = waitpid(pid, &status, 0);
} else if (pid == 0) {
 printf ("child here");
 …..
 exit(status);
} else {
  perror ("fork problem");
  exit (-1);
```

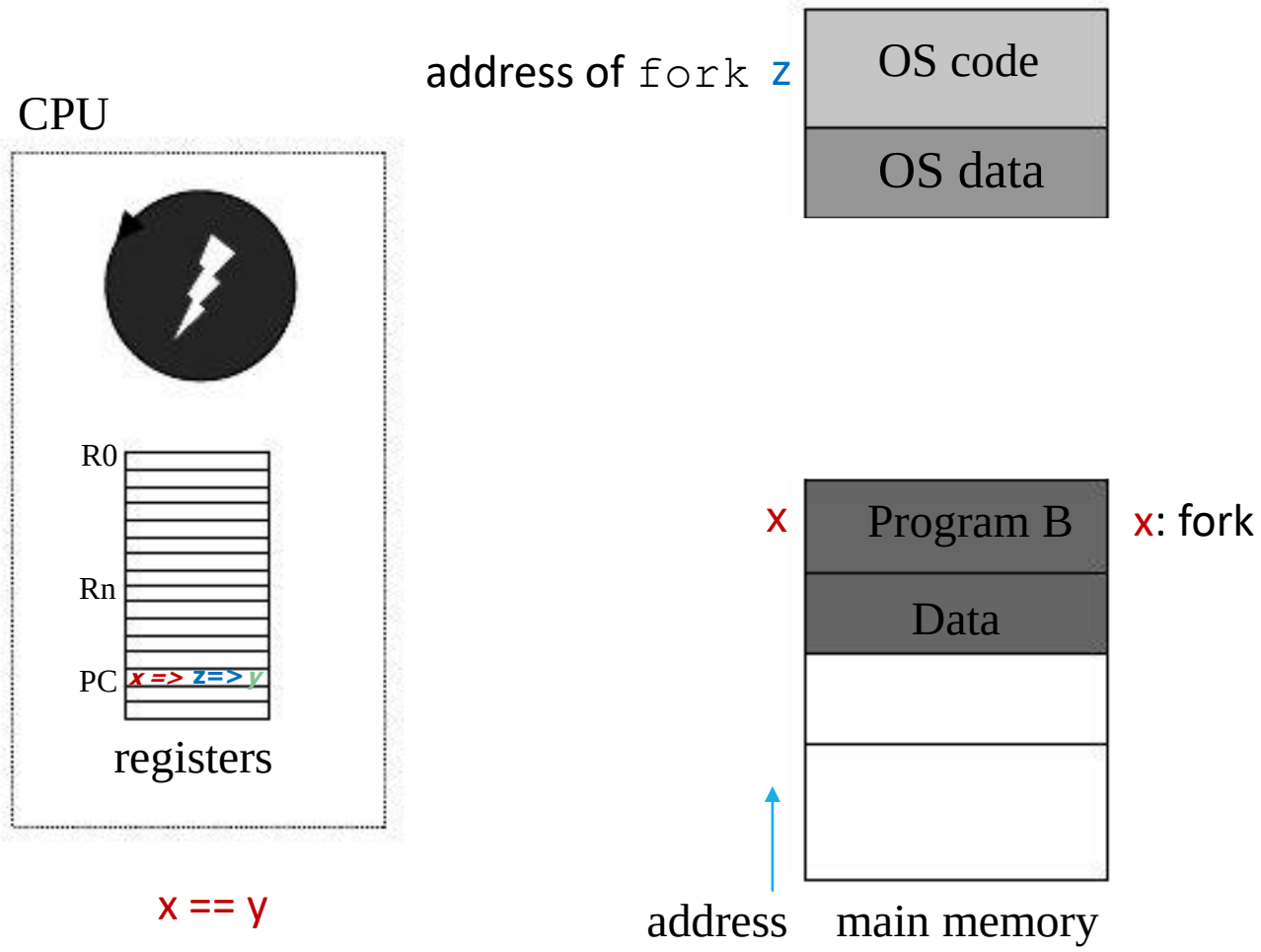
Entry point: `main`

CHILD

```
#include <unistd.h>
int pid;
int status = 0;
pid = fork ();
// HERE IS WHERE WE START, pid = 0
if (pid > 0) {
 printf ("Parent: child has
          pid=%d", pid);
         …..
 pid = waitpid(pid, &status, 0);
} else if (pid == 0) {
 printf ("child here");
 …..
 exit(status);
} else {
  perror ("fork problem");
  exit (-1);
```

Entry point: after `fork`

Running programs: memory and the CPU



Example: Process Creation via `Fork`

```
#include <unistd.h>
int pid;
int status = 0;
pid = fork ();
if (pid > 0) {
 printf ("Parent: child has
          pid=%d", pid);
         …
 pid = waitpid(pid, &status, 0);
} else if (pid == 0) {
  printf ("child here");
  …
  exit(status);
} else {
   perror ("fork problem");
   exit (-1);
```

`Fork` returns *twice*. It returns a 0 PID to the child and the child's PID to the parent

Parent typically **blocks** or `wait`s until the child terminates by using `wait` or `waitpid`

Child or any process can return an exit status

Always check for errors (-1) on syscalls

Fork example

simplefork.c:

- `while (1) fork ();`
  - fork “bomb”

Process Topologies

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(); // pid_t is a process id (#)
```

- simplefork.c
- simplechain.c

Waiting

Called by parent to `wait` (**block**) until child exits

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

`stat_loc`: {why} did child exit? (don't care, NULL or 0)
return parameter, if not NULL, must allocate it!

```
wait (NULL); // OR
int stat_loc;
wait (&stat_loc); // "output" param
```

What is in stat_loc? See 3.4.1 Status values (R&R)

1. status of child: exited, suspended, …
2. If normal exit, what was the exit value (i.e. child does `return (5)`);

Processes and shared variables

- Processes only share at CLONE time – copy

```
int i = 5;

childpid = fork();
if (childpid == 0)  {                          /* child code */
     print i;  //#1
     i = 7;
     print i;  //#2
 }
else {
    print i;      // #3                          /* parent code */
    i = 3;
    print i;      // #4
    wait (NULL);
    print i;      // #5
}
```

Top-down:Why Processes?

- Why we need fork? Multi-tab browser.

```
// #define MaxURL 100
const int MaxURL = 100;
int i, num_urls;
char *URLs [MaxURL];
pid_t child_pids [MaxURL];
// assume num_urls and URLs have been set
for (i=0; i<num_urls; i++)
        if ((child_pids[i] = fork()) == 0) {
                fetch_and_display (URLs[i]);
                break;
        }
for (i=0; i<num_urls; i++) {
        wait (NULL);
        // why wait? cleanup any state related to tabs
}
```

ISOLATION!
for tabs

How can Fork fail?

Too many processes in the system

Not enough memory

Not enough disk space (later)

Back to `Fork` (cont'd)

- The child process may execute a different *program*: `exec*` call

- Still need `fork` to create the child clone and container, but `exec*` changes the code it runs

- It completely over-writes the child's memory
  - Except: IDs (pid) are intact as are I/O descriptors, environment

Why do we need exec?

// implementation of shell

```
>./foo
>gcc …
```

// implementation of make

```
foo.o: foo.h foo.c
    gcc foo.c
    cp foo.o /usr/bin/foo.o
    …
```

execl

```
int execl (const char *path,
     const char *arg0,  // strings
     const char *arg1, …
     (char *) 0);
```

Executes program named by 1st argument
Pathname+executable name (e.g. "`/usr/jon/prog`")

`arg0` is just the name of the executable "`prog`"
remainder are optional, with 0 or NULL terminating

Many Options

- Family of options
  - `execl, execlp, execv, execvp, execve`
  - Differ in how arguments + env are used

- All short forms of execve

```
int execve (const char *path,
            const char *argv[],
            const char *envp[]);
```

provides new values for env

Example:

`<execcmd.c>`

How can `exec*` fail?

Why do we need `exec*`: Consider the Shell

```
while (TRUE) {                                  /* repeat forever */
  type_prompt( );                               /* display prompt */
  read_command (command, parameters)            /* input from terminal */
  /* some error checking … */
if (fork() != 0) {                              /* fork off child process */
  /* Parent code */
  waitpid( -1, &status, 0);                     /* wait for any child to exit */
} else {
  /* Child code */
  execve (command, parameters, 0);              /* execute command */
 }
}
```

If command crashes, no problem!

Shell with `execve`

- When you run your program:

shell> `my_prog 0 10`

The Shell is doing the `fork/exec`

path/command is *current working directory* + `my_prog`

`argv:`

`arg0` is "`my_prog`"

`arg1` is "`0`"

`arg2` is "`10`"

Questions

```
read_command (command, parameters)
if (fork() != 0) {
   /* Parent code */
   waitpid( -1, &status, 0);
} else {
  execve (command, parameters, 0);
}
}
```

How would you implement:

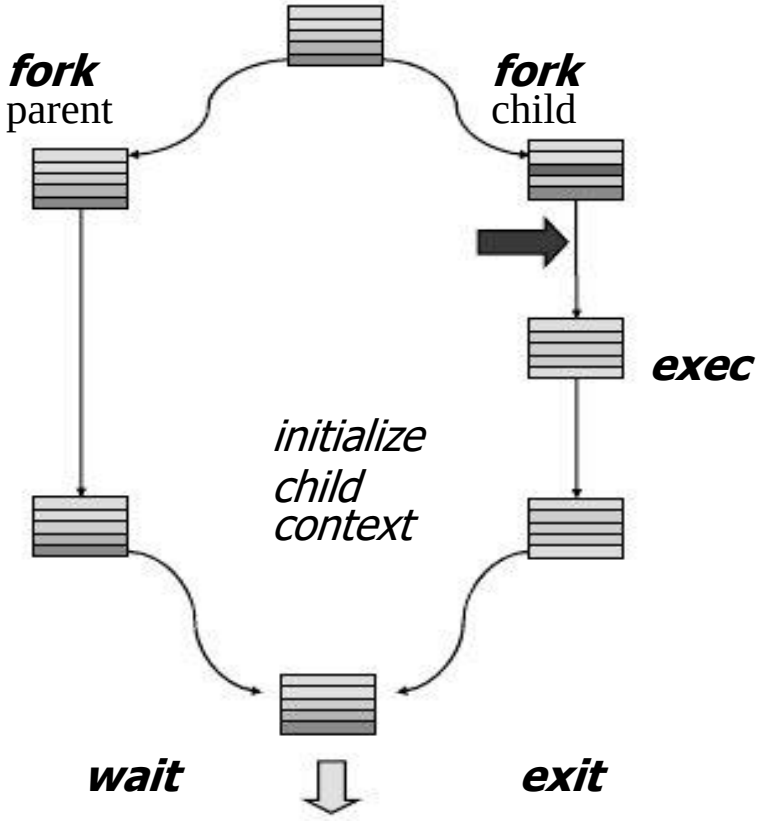
1) shell> my_prog&

2) what would happen if the shell did not fork, just exec?

Uses of exec*

- Program that runs other programs

Putting it all together



Process Termination

- How does a process terminate?
  - Return from `main`
  - Falls off the end of `main`
  - Call `exit`
  - Call `abort`
  - Receives a death signal or exception

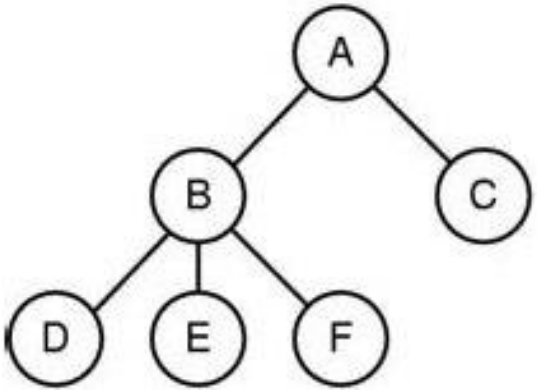
```
kill (pid_t pid, SIGKILL)
```

Process Termination (cont'd)

- Should be orderly
  - Child terminates before parent and parent is `wait`ing
- If child `exit`s when parent is not `wait`ing
  - Child becomes a "zombie"
  - OS keeps it around long enough until parent does a `wait` to get `exit` status or until parent exits

Process Termination (cont'd)

- Parent exits while child is running
  - Child becomes an orphan
  - No problem as children become "adopted" by parent up the tree, possibly *systemd*



Identities

- When a process runs, OS must associate a user and group id to it, why?
  - accounting and security

`fork` () returns pid of the child to the parent

`getpid` () returns the pid of the calling process

`getppid` () returns my parents pid

`getuid` () returns userid of the user that started the process (e.g. “jon” => 89392)

Shell and &

- Does the shell ever wait for background processes?

- Yes, it receives a signal (SIGCHLD) when child exists (to be discussed later)

Next Time

- Begin discussion of input/output
- Read Chapter 4 R&R
- Project #1 will be available on Weds