

Processes (part 1)

M1 MOSIG – Operating System Design

Renaud Lachaize

Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
 - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
 - David Mazières (Stanford)
 - (many slides/figures directly adapted from those of the CS140 class)
 - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
 - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)
 - Textbook: Computer Systems: A Programmer's Perspective (2nd Edition) a.k.a. "CSAPP"
 - CS 15-213/18-243 classes (many slides/figures directly adapted from these classes)
 - Textbooks (Silberschatz et al., Tanenbaum)

Outline

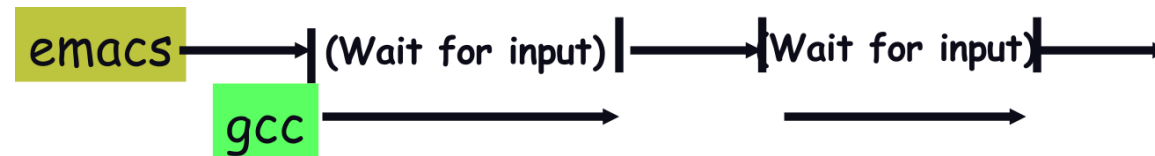
- Introduction
- Basic process management interface from the user-level
- Kernel-level process management

Processes

- A process is an instance of a running program
- Modern OSes run multiple programs simultaneously
- Examples (can all run simultaneously)
 - `gcc file1.c` – compiler running on file 1
 - `gcc file2.c` – compiler running on file 2
 - `emacs` – text editor
 - `firefox` – web browser
- Non-examples (implemented as one process):
 - Multiple `firefox` windows or `emacs` frames (still one process)
- Why processes?
 - Simplicity of programming
 - Higher CPU throughput, lower latency (details on next slide)

Speed

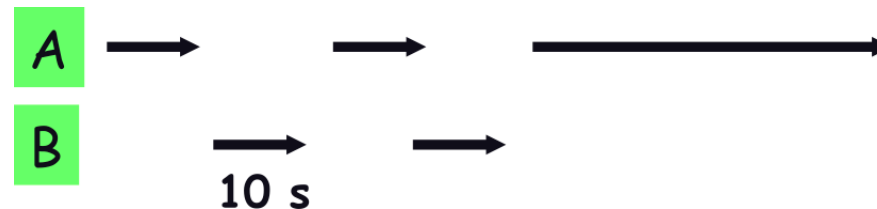
- Multiple processes can increase CPU utilization
 - Overlap one process with another's wait



- Multiple processes can reduce latency
 - Running *A* then *B* requires 100 sec for *B* to complete



- Running *A* and *B* concurrently makes *B* finish faster



A process's view of the world

- Each process has its own view of the machine
 - Its own address space
 - Its own open files
 - Its own virtual CPU (through preemptive multitasking)
- A given (virtual) address has a different “meaning” for two distinct processes
- This greatly simplifies the application programming model
 - `gcc` does not care that `firefox` is running
- Sometimes interaction between processes is necessary
 - Simplest is through files: `emacs` edits file, `gcc` compiles it
 - More complicated: Shell/command, Window manager/application

Outline

- Introduction
- Basic process management interface from the user-level
- Kernel-level process management

Creating a new process

- **pid_t fork(void)**
 - Creates a new process (child process) that is identical to the calling process (parent process)
 - Returns 0 to the child process
 - Returns the child's pid to the parent process

```
pid_t pid = fork();  
if (pid == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

- **fork** is interesting (and often confusing) because it is called once but returns twice

Understanding fork

Process n

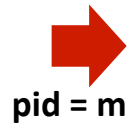


```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

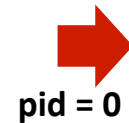
Child Process m



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

hello from parent

Which one is first?

hello from child

Fork: additional details

- Parent and child both run the same code
 - Distinguish parent from child by return value from `fork`
- Start with the same state, but each has private copy
 - Memory address space
 - Environment variables
 - List of currently open files
 - Signal handlers, signal mask and list of pending signals
- Scheduling non-determinism
 - Who runs first after a fork? Parent? Child? (Both?)
 - No order imposed/specified

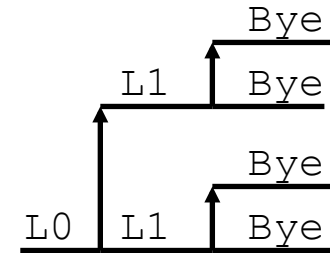
Fork Example #1

```
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);
}
```

Fork Example #2

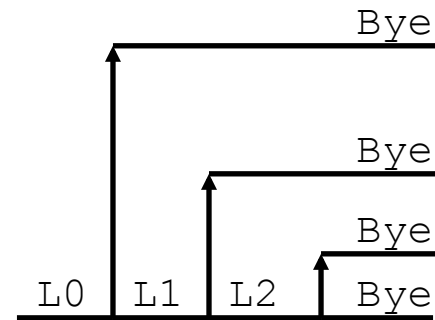
- Both parent and child can continue forking

```
void fork2()  
{  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("Bye\n");  
}
```



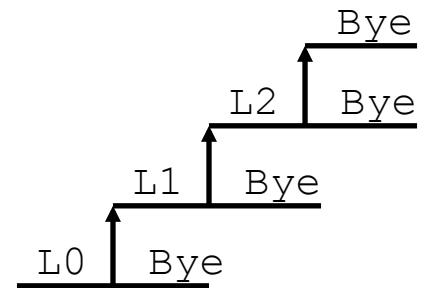
Fork example #3

```
void fork4()  
{  
    printf("L0\n");  
    if (fork() != 0) {  
        printf("L1\n");  
        if (fork() != 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```



Fork example #4

```
void fork5()  
{  
    printf("L0\n");  
    if (fork() == 0) {  
        printf("L1\n");  
        if (fork() == 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```



Ending a process

- **void exit(int status)**
 - Exits a process (current process ceases to exist)
 - By convention, status of 0 is success (non-zero means error)
 - **atexit()** allows programmer to register function to be executed upon exit

```
void cleanup(void) {  
    printf("cleaning up\n");  
    ... // clean things up  
    printf("done\n");  
}  
  
int main() {  
    atexit(cleanup);  
    ... // do useful things  
    exit(0);  
}
```

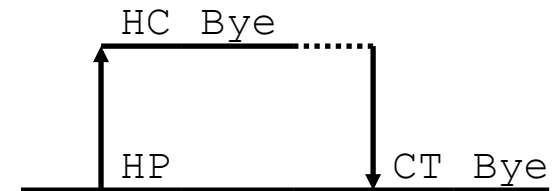
Synchronizing with child processes

- `int wait(int *child_status)`
 - suspends current process until one of its children terminates
 - return value is the `pid` of the child process that terminated
 - if `child_status != NULL`, then the variable it points to will be set to a status indicating why the child process terminated

Synchronizing with child processes

`wait`: example #1

```
int main() {  
    int child_status;  
  
    if (fork() == 0) {  
        printf("HC: hello from child\n");  
    }  
    else {  
        printf("HP: hello from parent\n");  
        wait(&child_status);  
        printf("CT: child has terminated\n");  
    }  
    printf("Bye\n");  
    exit();  
}
```



Synchronizing with child processes

`wait`: details and example #2

- If multiple children completed, will take in arbitrary order
- Can use macros **WIFEXITED** and **WEXITSTATUS** to get information about exit status (see `man 2 wait` for details)

```
int main()
{
    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 terminate abnormally\n", wpid);
    }
}
```

Synchronizing with child processes

Waiting for a specific process

- **waitpid(pid, &status, options)**
 - suspends current process until specific process terminates
 - various options – see man page for details

```
int main()
{
    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)) /* Child terminated due to call to exit */
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Zombies

- Idea
 - When a process terminates, it still consumes system resources
 - Various tables maintained by OS
 - Called a “zombie”
 - Living corpse, half alive and half dead
- Reaping
 - Performed by parent on terminated child via `wait` or `waitpid`
 - Parent is given exit status information
 - Kernel discards process
- What if parent doesn't reap?
 - If any parent terminates without reaping a child, then child will be reaped by a special OS process (often called `init`)
 - So, we only need explicit reaping in long-running processes
 - e.g., shells and servers

Zombies – Example

```
linux> ./forks &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6639 ttyp9        00:00:03 forks
 6640 ttyp9        00:00:00 forks <defunct>
 6641 ttyp9        00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6642 ttyp9        00:00:00 ps
```

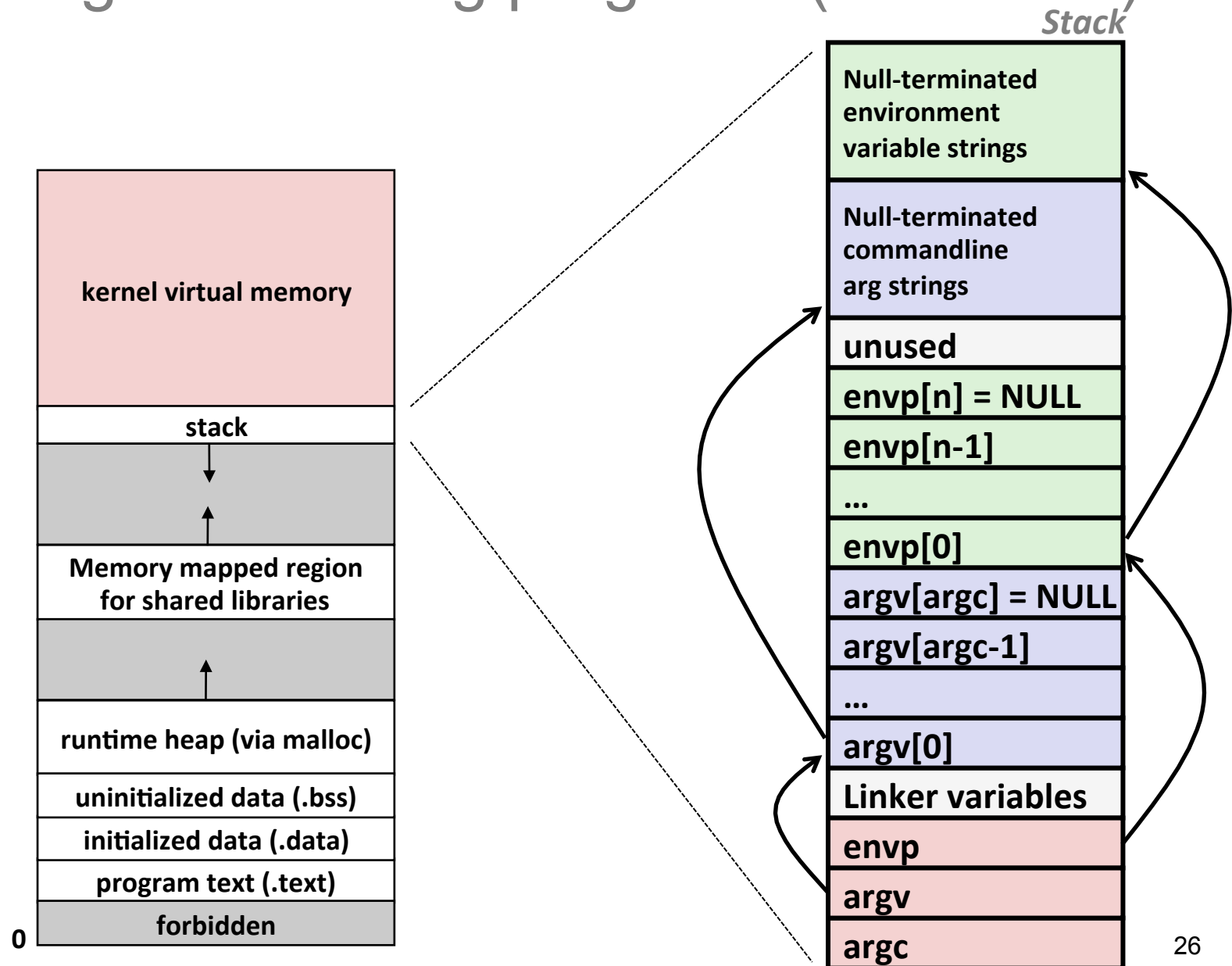
```
int main()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
            getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
            getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

- `ps` shows child process as “defunct”
- Killing parent allows child to be reaped by `init`

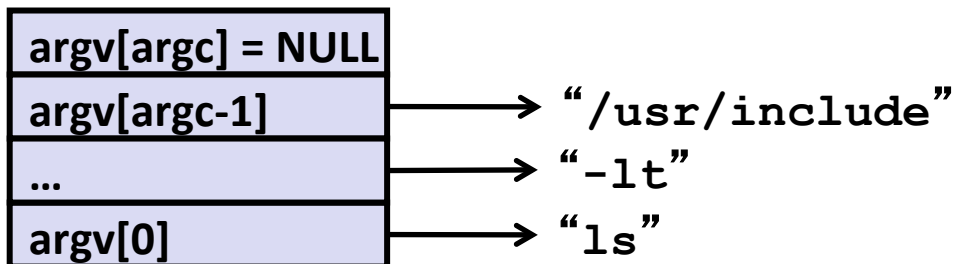
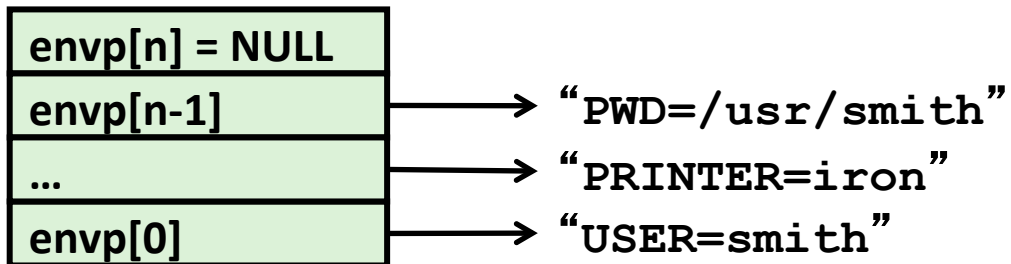
Loading and running programs

- `int execve(char *filename, char *argv[], char *envp);`
- Completely reconfigures the image of the calling process
- Loads and runs
 - Executable `filename`
 - With argument list `argv` (NULL terminated)
 - By convention, `argv0` should be the name of the executable file
 - And environment variable list `envp`
- **Does not return!** (unless error)
- Overwrites process memory address space, keeps pid
- Environment variables
 - “name=value” strings

Loading and running programs (continued)



Execve arguments: example



exec1 and exec family

- `int exec1(char *path, char *arg0, char *arg1, ..., 0)`
- Loads and runs executable at path with args `arg0, arg1, ...`
 - `path` is the complete path of an executable object file
 - By convention, `arg0` is the executable object file
 - “Real” arguments to the program start with `arg1`, etc.
 - **List of args is terminated by a `(char*) 0` argument**
 - Environment (implicitly) taken from `char **environ` (an automatically defined global variable), which points to an array of “name=value” strings

`exec1` and `exec` family (continued)

- Family of functions includes `execv`, `execve`, `execvp`, `exec1`, `execle`, `execlp`
- **These functions only differ in terms of interface; their purpose is the same (loading and running a new process, in the context of the current process)**
- Usually, `execve` is the only system call
 - The other functions are implemented as library wrappers
- **Semantics of the suffixes**
 - **v** (vector): pass arguments as (**NULL**-terminated) array of pointers
 - **l** (list): pass arguments as (**NULL**-terminated) list of pointers (i.e., the function has a variable number of arguments)
 - **p** (path): search executable file name in the **PATH** (otherwise, full path must be provided)
 - **e** (environment): explicitly provide new environment (otherwise, use the current environment pointed by `environ`)

exec1: example

```
int main() {
    int res;
    if (fork() == 0) {
        res = exec1("/usr/bin/cp", "cp", "foo", "bar", 0);
        if (res < 0) {
            printf("error: exec1 failed\n");
            exit(-1);
        }
    }
    wait(&res);
    if (res == 0) {
        printf("copy completed\n");
    }
    exit(0);
}
```

Wrap-up: mini-shell example

- A *shell* is an application program that runs programs on behalf of the user
 - Interpreted shell scripts (not detailed here)
 - Built-in commands (e.g., `cd`)
 - External binary programs
- Examples: `tcsh`, `bash`

```
int main()
{
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("myshell> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

*Execution is a sequence of
read/evaluate steps*

Wrap-up: mini-shell example (continued)

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;               /* should the job run in bg or fg? */
    pid_t pid;            /* process id */

    bg = parseline(cmdline, argv);
    /* builtin_command handles internal commands such as 'cd' */
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) { /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) { /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

What is a “background job”?

- Users generally run one command at a time
 - Type command, read output, type another command
- Some programs run “for a long time”
 - Example: “delete this file in two hours”
`% sleep 7200; rm /tmp/junk # shell stuck for 2 hours`
- A “background” job is a process we don't want to wait for
`% (sleep 7200 ; rm /tmp/junk) &`
`[1] 907`
`% # ready for next command`

Problem with mini-shell example

- Shell correctly waits for and reaps foreground jobs
- But what about background jobs?
 - Will become zombies when they terminate
 - Will never be reaped because shell (typically) will not terminate
 - Will create a memory leak that could theoretically run the kernel out of memory
 - Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: `fork()` returns -1
- Solution: asynchronously notify the shell when a child process terminates
 - Using “Unix signals” (details on this mechanism soon)

Outline

- Introduction
- Basic process management interface from the user-level
- Kernel-level process management

Implementing processes

- The OS kernel keeps a data structure for each process
 - Generic name: Process control block (PCB)
 - Sometimes named differently, e.g., `task_struct` in Linux
- Tracks states of the process
 - Running, ready (runnable), blocked, etc.
- Includes information necessary to run
 - Saved context (registers ...)
 - Information about virtual memory (memory region descriptors, pointer to page tables ...)
 - Information about open files and memory-mapped files
- Various other data about the process
 - Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, ...

Fork revisited – How does it work internally?

- Create and initialize kernel data structures about the new process, such as:
 - Process control block (PCB) and corresponding pid
 - Open file descriptors and memory mapped files
 - Inherited from parent – each process can close them independently
- Create and initialize virtual memory address space
 - For efficiency, the whole address space of the parent process is not copied
 - Instead, only copy memory region descriptors and page tables
 - Then, use Copy-on-Write (CoW) to track and support changes between parent and child address spaces
 - Mark PTEs of writeable regions as read-only
 - Flag region descriptors for these areas as private “copy-on-write”
 - Writes by either process to these pages will cause page faults
 - Fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions

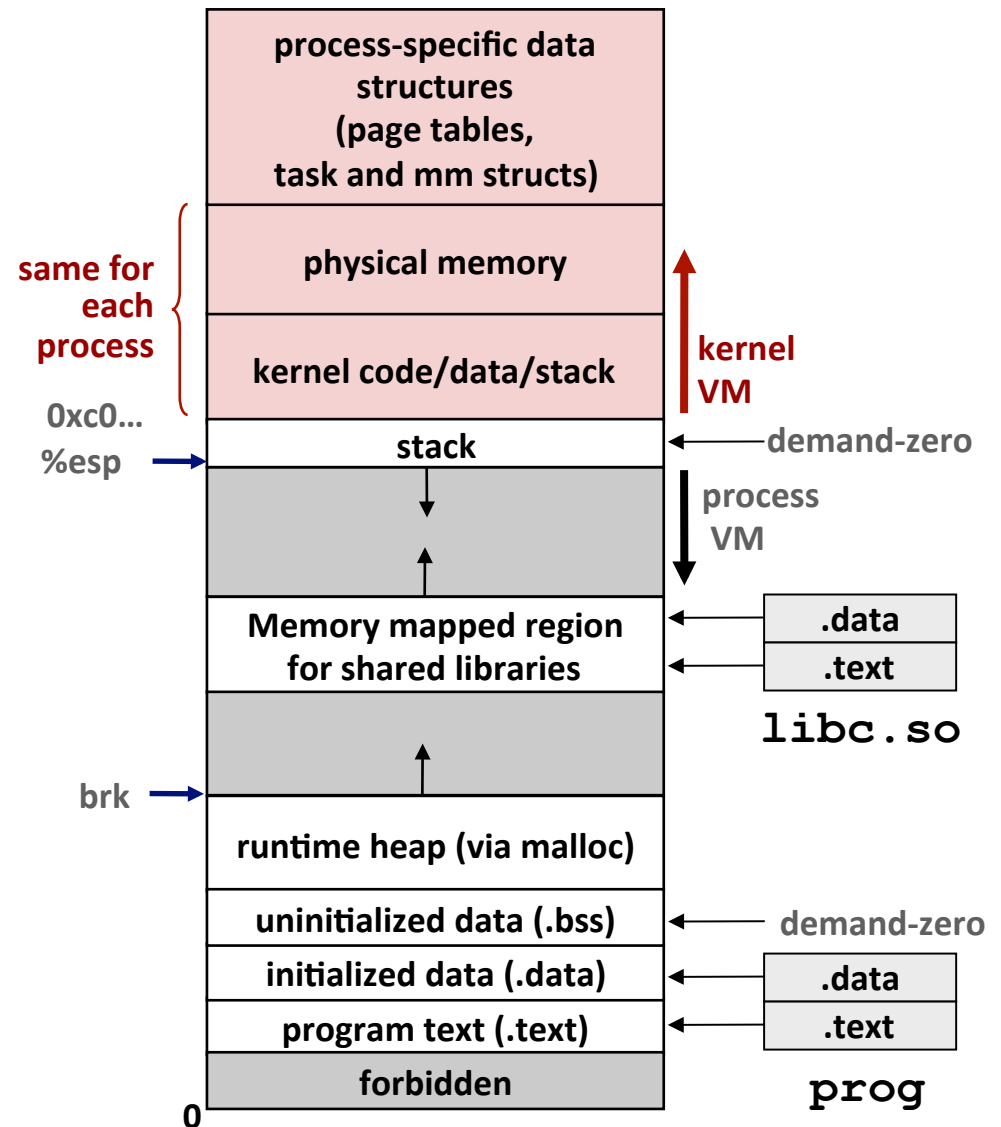
Fork revisited – How does it work internally? (continued)

- Conclusion
 - Thanks to CoW, copies of memory pages are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page)
 - Important optimization, especially given the fact that **fork** is often followed by a call to **exec**, which completely reconfigures the contents of the virtual memory address space

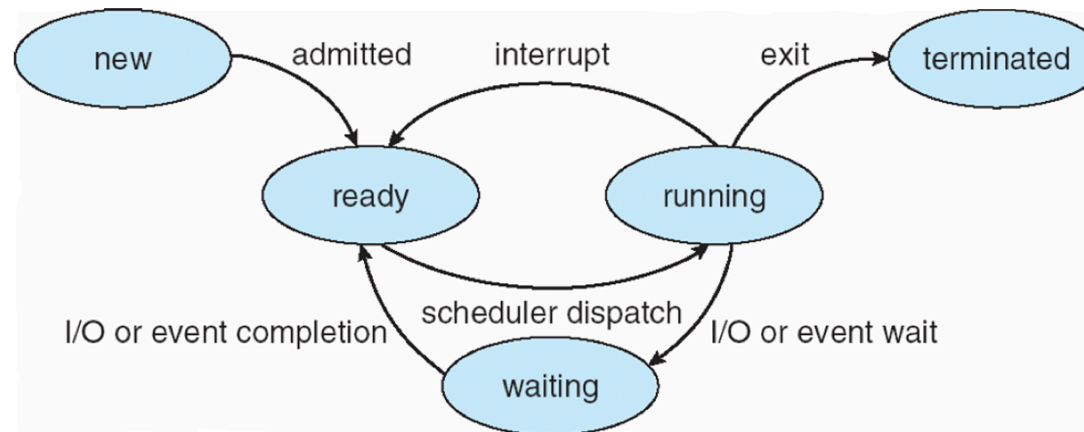
exec revisited

To run a new program **prog** in the current process using **exec()** :

- Free the region descriptors and page tables for the existing memory regions
- Create new region descriptors and page tables according to the contents of the new executable file
 - Stack, BSS, data, text, shared libs.
 - Text and data backed by ELF executable object file
 - BSS and stack initialized to zero
 - Shared libraries (e.g., **libc.so**)
- Set PC to entry point to main in **.text**
 - The kernel will fault in code, data pages as needed

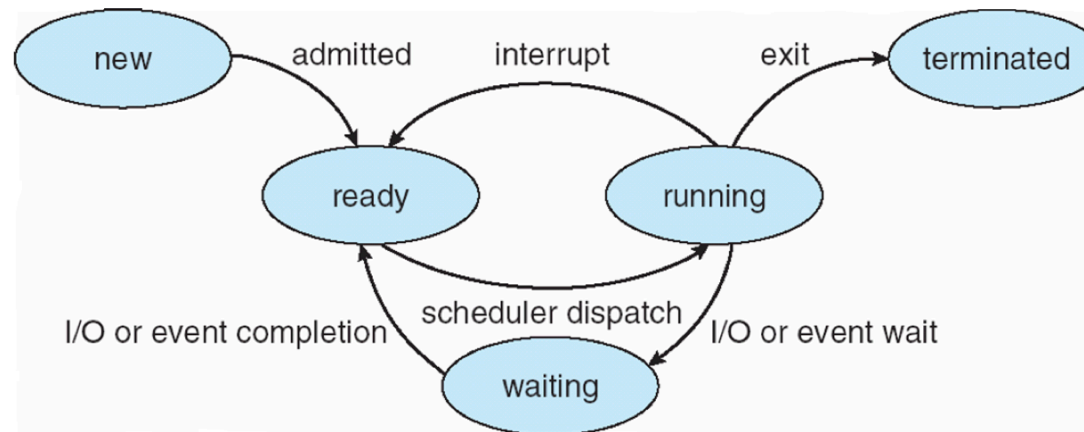


Process states



- A process can be in one of several states
 - New/terminated: at beginning/end of life
 - Running – currently executing (or will execute on kernel return)
 - Ready – can run, but kernel has chosen different process to run
 - Waiting – needs external event (e.g., end of disk operation) to proceed
- Which process should the kernel run?
 - If non runnable, run idle loop, if a single process runnable, run this one
 - If more than one runnable process, must make scheduling decision

CPU scheduling



- Scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from new/waiting to ready
 4. Exits
- Non-preemptive schedules use only 1 & 4
- Preemptive schedulers run at all four points

Scheduling criteria

- A few important criteria
 - **Throughput**: Number of processes that complete per time unit (higher is better)
 - **Turnaround time**: Time for each process to complete (lower is better)
 - **Response time**: time from request to first response (e.g., key press to character echo) (lower is better)
- The above criteria are affected by secondary criteria
 - **CPU utilization**: fraction of time that the CPU spends doing productive work (i.e., not idle)
 - **Waiting time**: time that each process spends waiting in ready queue

Scheduling

- How to pick which process to run?
- Scan process table for first runnable?
 - Expensive. Weird priorities (small pids better)
 - Divide into runnable and blocked processes
- FIFO?
 - Put process on back of list, pull them off from front



- Priority?
 - Give some processes a better shot at the CPU

Scheduling policy

- Want to balance multiple goals. For example:
 - *Fairness*: don't starve processes
 - *Priority*: reflect relative importance of processes
 - *Deadlines*: must do x (e.g., play audio) by certain time
 - *Reactivity*: minimize response time
 - *Throughput* : want good overall performance (e.g., number of processes completed per time unit)
 - *Efficiency*: minimize overhead of scheduler itself
- No universal policy
 - Many objectives – cannot optimize for all
 - Conflicting goals (e.g., throughput or priority versus fairness)

Preemption

- **A process can be preempted when kernel gets control – There are several such opportunities:**
- **A running process can transfer control to kernel through a trap**
 - System call (including `exit`), page fault, illegal instruction, etc.
 - May put current process to sleep – e.g., read from disk
 - May make other process runnable – e.g., `fork`, write to pipe
 - (May destroy current process)
- **Periodic timer interrupt**
 - If running process used up time quantum, schedule another
- **Device interrupt**
 - E.g., disk request completed, or packet arrived on network
 - A previously waiting process becomes runnable
 - Schedule if higher priority than current running process
- Recall that changing the running process is called a **context switch**

Context switch details

- Implementation is very machine (processor) dependent. Typical things include:
 - Save/restore general registers
 - Save/restore floating point or other special registers
 - Switch virtual address translations (e.g., pointer to root of paging structure)
 - Save/restore condition codes (flags)
 - Save/restore program counter
- Non-negligible cost
 - Save/restore of floating point registers is expensive
 - Optimization: only saved if process used floating point operations
 - May require flushing TLB (memory translation hardware)
 - Optimization: do not flush the kernel's own (global) data from the TLB
 - Usually causes L1/L2/L3 cache misses (switches working sets)