

Operating Systems

04. Processes

Paul Krzyzanowski

Rutgers University

Spring 2015

Key concepts from last week

Boot Loader

- Multi-stage boot loader
- Old Intel PC architecture (*still used!*)
 - BIOS
 - Master Boot Record – located at block 0
 - Volume Boot Record
 - OS Loader
- Current PC architecture (2005+)
 - UEFI – knows how to read one or more file systems
 - Loads OS loader from a boot partition
- Embedded systems (e.g., ARM-based devices)
 - Custom boot firmware on the processor chip

Operating System vs. Kernel

- Kernel
 - “nucleus” of the OS; main component
 - Provides abstraction layer to underlying hardware
 - Manages system resources (CPU, file systems, memory, network)
 - Enforces policies
- Rest of the OS
 - Utility software, windowing system, print spoolers, etc.
- Kernel mode vs. user mode execution
 - Flag in the CPU
 - Kernel mode = can execute privileged instructions

Mode switch

- Transition from user to kernel mode (and back)
- Includes a change in flow
 - Cannot just execute user's instructions in kernel mode!
 - Well-defined addresses set up at initialization
- Change mode via:
 - Hardware interrupt
 - Software trap (or syscall)
 - Violations (exceptions): illegal instruction or memory reference

Context Switch

- Mode switch + change executing process

Timer interrupts

- Crucial for:
 - Preempting a running process to give someone else a chance (force a context switch)
 - Including ability to kill the process
 - Giving the OS a chance to poll hardware
 - OS bookkeeping

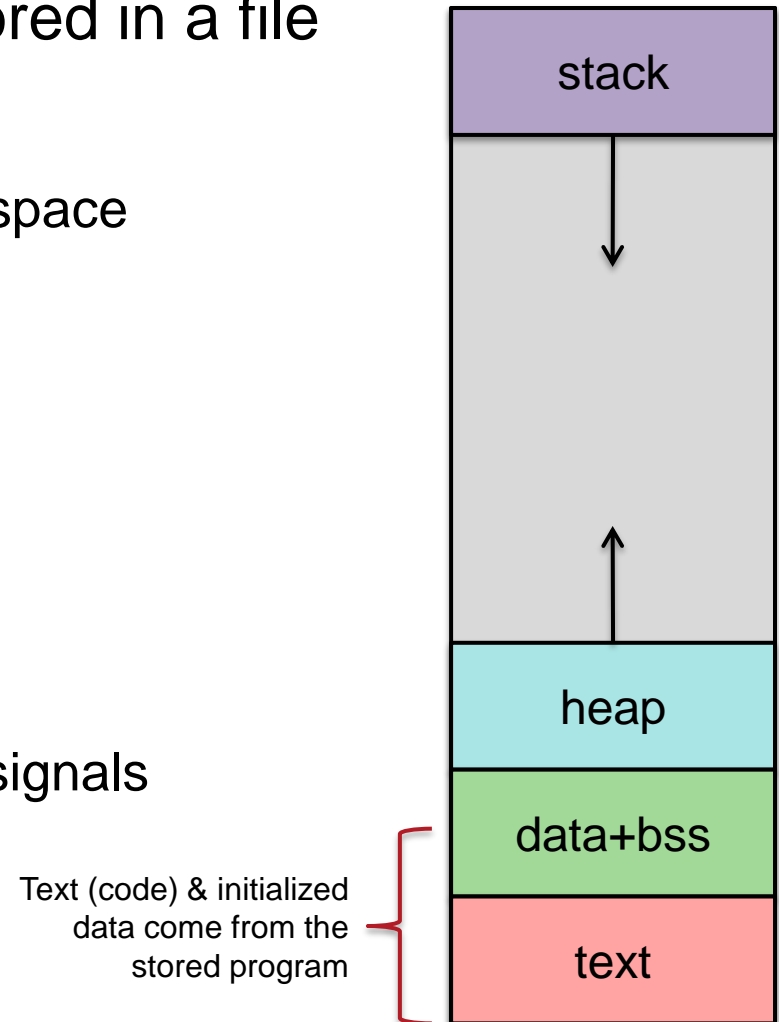
Timer interrupts

- Windows
 - Typically 64 or 100 interrupts per second
 - Apps can raise this to 1024 interrupts per second
- Linux
 - Interrupts from Programmable Interval Timer (PIT) or HPET (High Precision Event Timer) and from a local APIC timer (one per CPU)
 - all at the same rate
 - Interrupt frequency varies per kernel and configuration
 - Linux 2.4: 100 Hz
 - Linux 2.6.0 – 2.6.13: 1000 Hz
 - Linux 2.6.14+ : 250 Hz
 - Linux 2.6.18 and beyond: aperiodic – **tickless kernel**
 - PIT not used for periodic interrupts; just APIC timer interrupts
 - Kernel determines when the next interrupt should take place

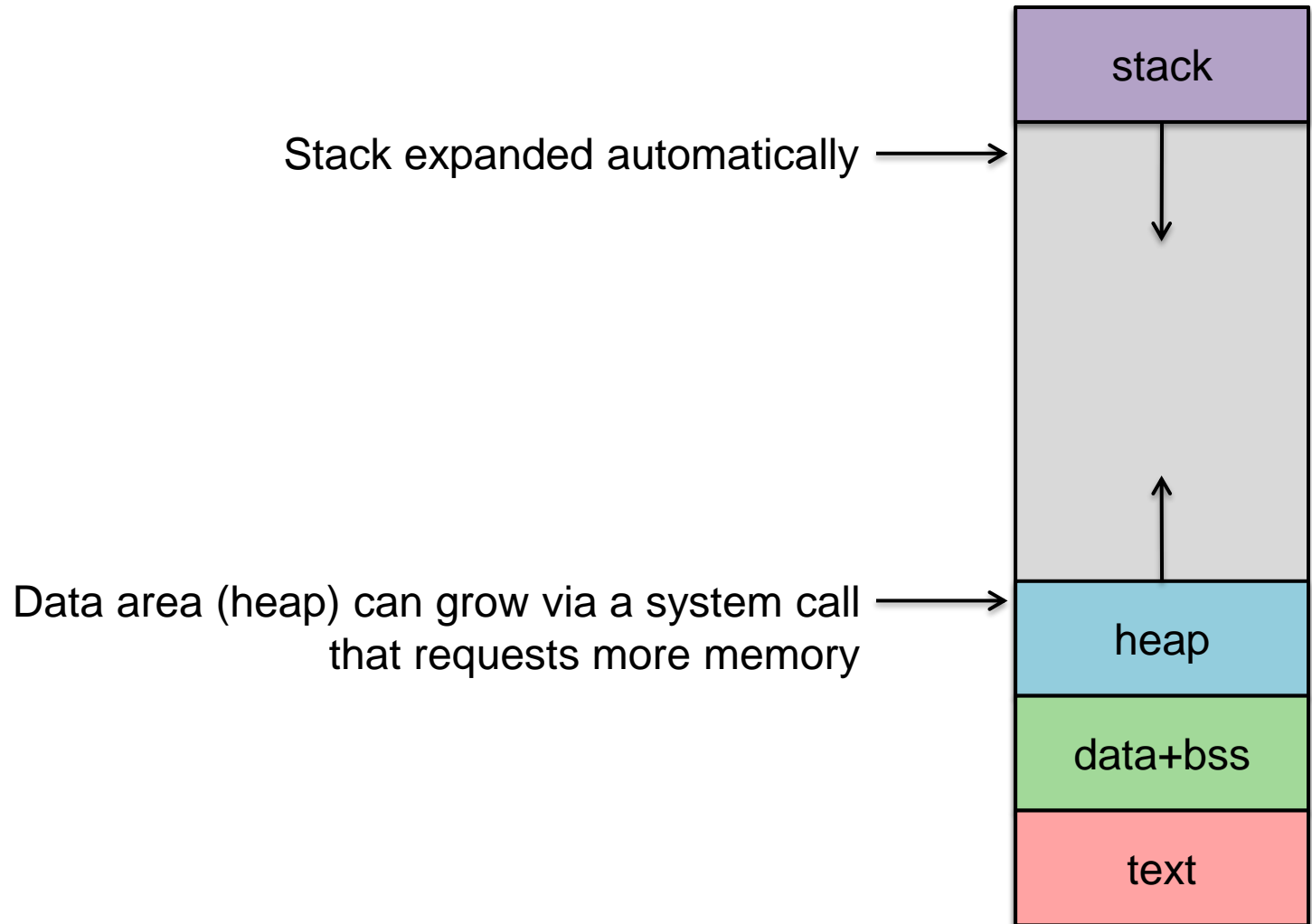
Processes

Process

- **Program**: code & static data stored in a file
- **Process**: a running program
 - Each process has its own address space (we'll look at this later)
 - **Memory map**
 - **Text**: compiled program
 - **Data**: initialized static data
 - **BSS**: uninitialized static data
 - **Heap**: dynamically allocated memory
 - **Stack**: call stack
 - **System state**: open files, pending signals
 - **Processor state**:
 - Program counter
 - CPU registers



Growing memory



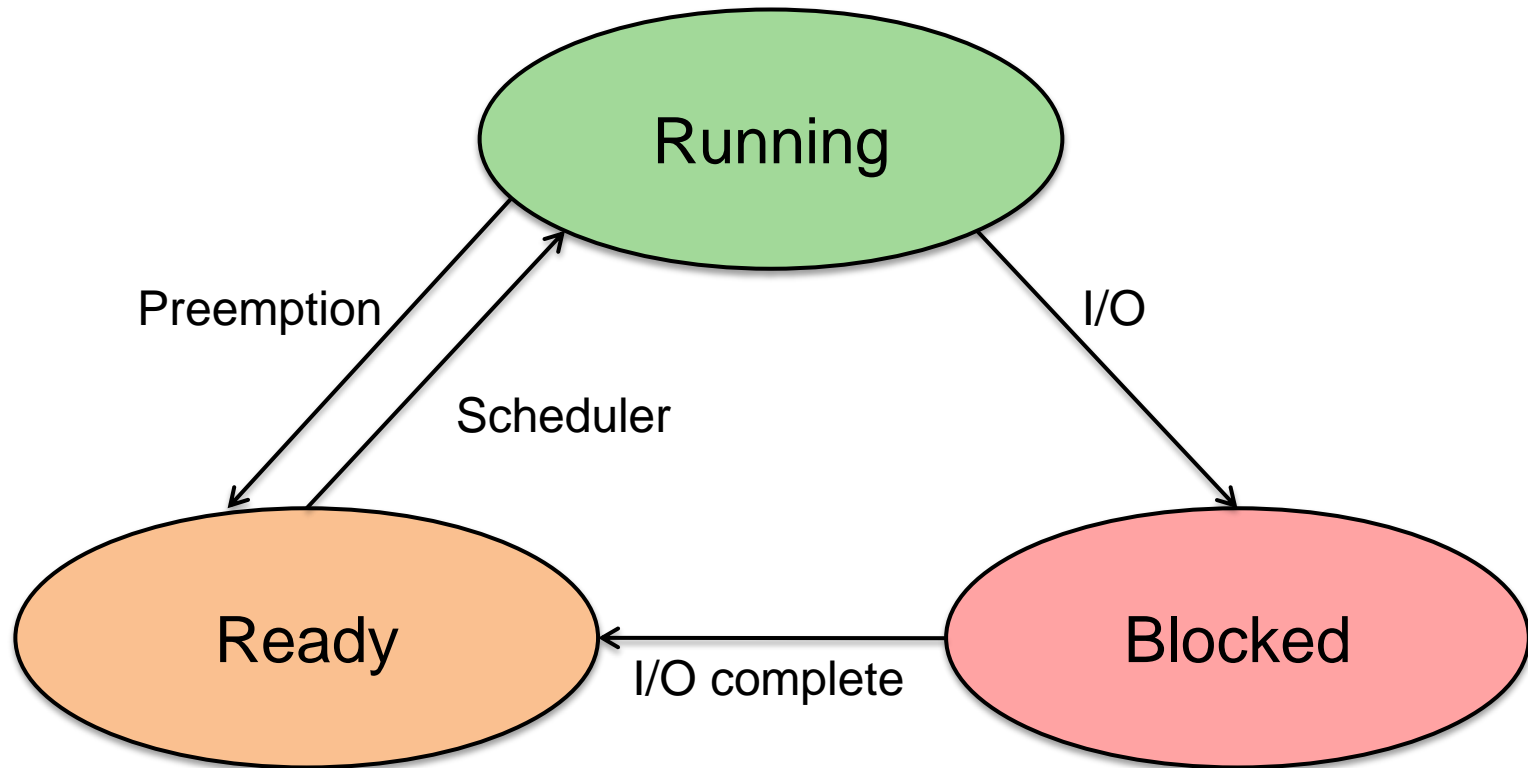
Contexts

- Entering the kernel
 - **Hardware interrupts**
 - Asynchronous events (I/O, clock, etc.)
 - Do not relate to the context of the current process [**kernel context**]
 - **Violations**
 - Are related to the context of the current process [**process context**]
 - Examples: illegal memory access, divide by zero, illegal instruction
 - **Software initiated traps (software interrupts)**
 - System call from the current process [**process context**]
- *The view of memory does not change on a trap*
 - *The currently executing process' address space is active on a trap*
- Saving state
 - Kernel stack switched in upon entering kernel mode
 - Kernel must save machine state before servicing event
 - Registers, flags (program status word), program counter, ...

Processes in a Multitasking Environment

- Multiple concurrent processes
 - Each process has a unique identifier: **Process ID** (PID)
- Asynchronous events (interrupts) may occur
 - The OS will have to take care of them
- Processes may request operations that take a long time
 - They have nothing to do now but wait
- Goal: **always have some process running**
 - Context saving/switching
 - Processes may be suspended and resumed
 - Need to save all state about a process so we can restore it

Process states



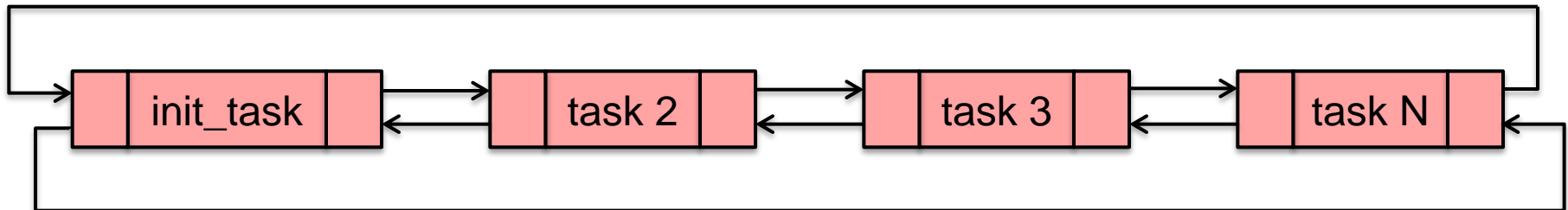
Keeping track of processes

- Process list stores a **Process Control Block (PCB)** per process
- A Process Control Block contains:
 - Process ID
 - Machine state (registers, program counter, stack pointer)
 - Parent & list of children
 - Process state (ready, running, blocked)
 - Memory map
 - Open file descriptors
 - Owner (user ID) – determine access & signaling privileges
 - Event descriptor if the process is blocked
 - Signals that have not yet been handled
 - Policy items: Scheduling parameters, memory limits
 - Timers for accounting (time & resource utilization)
 - (Process group)

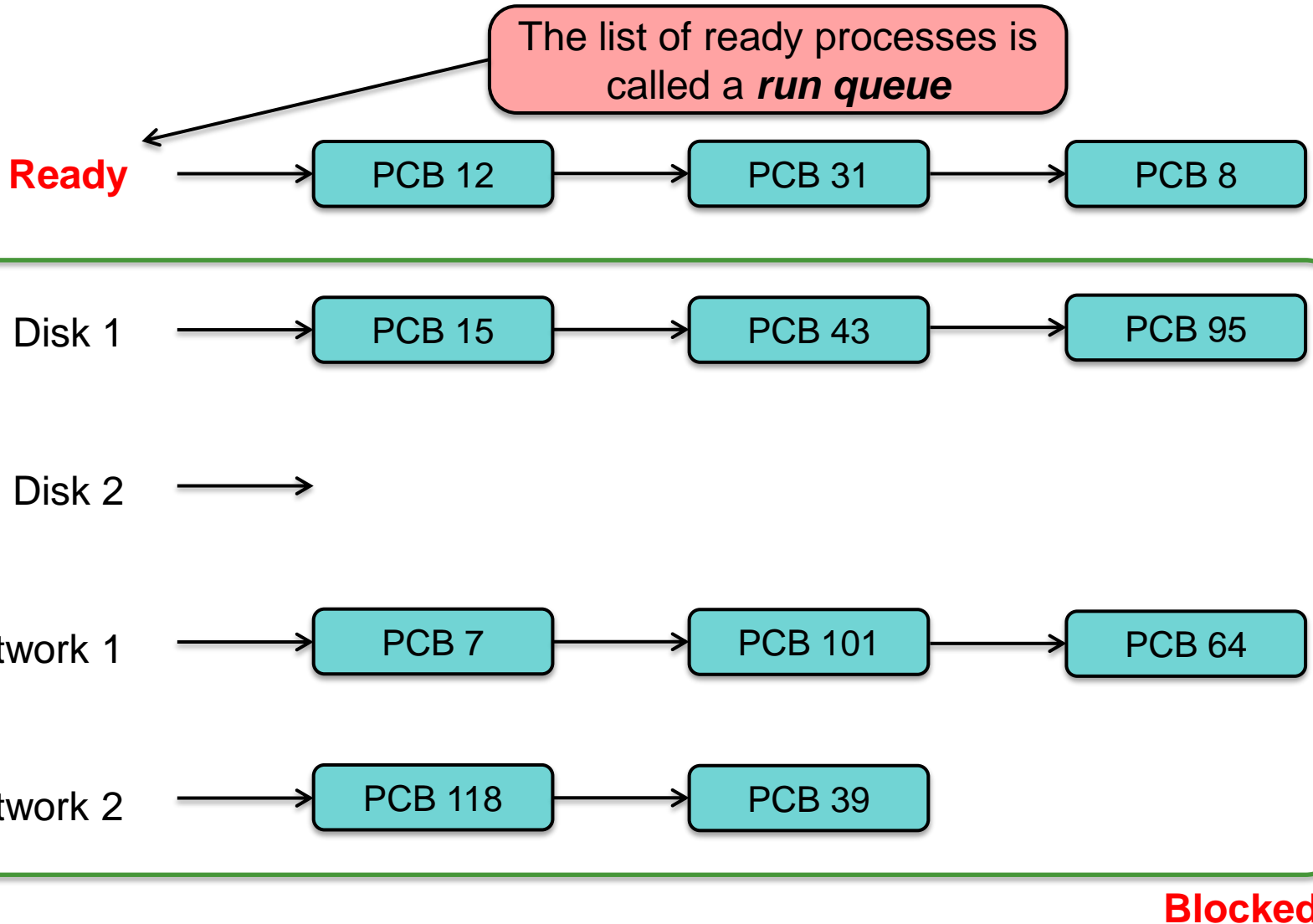
Processes in Linux

- The OS creates one task on startup:
 - init*: the parent of all tasks
 - launchd*: replacement for *init* on Mac OS X and FreeBSD
- Process state stored in `struct task_struct`
 - Defined in `linux/sched.h`
- Stored as a circular, doubly linked list

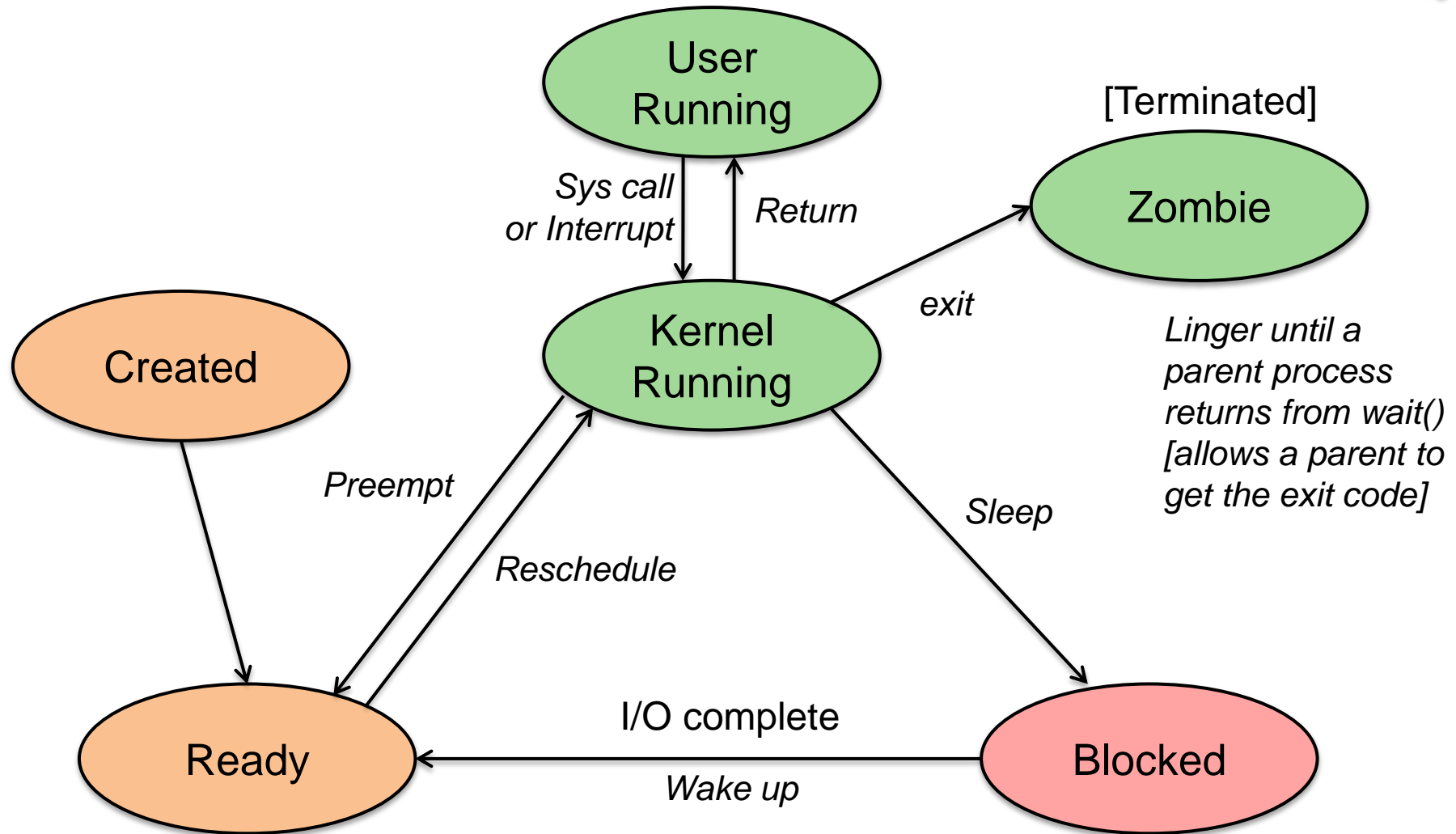
```
struct task_struct init_task;    /* static definition of the first task*/
```



Processes on Ready & Blocked Queues



Process States: a bit more detail

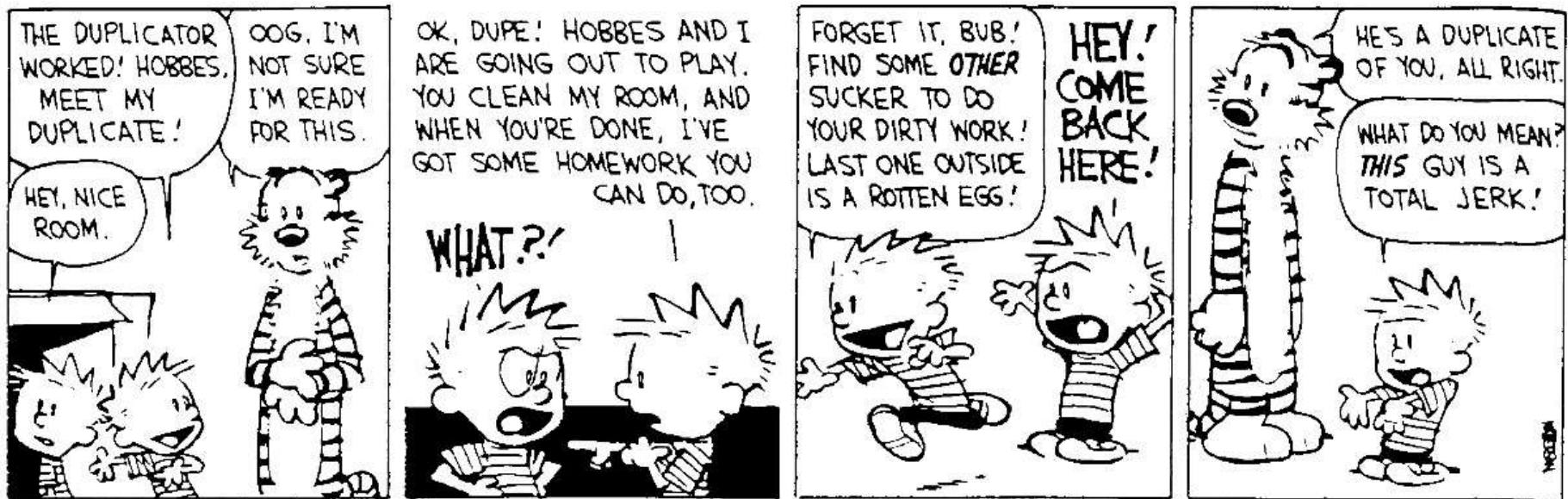


interrupt context: not part of any process state

Creating a process under POSIX

fork system call

- Clones a process into two processes
 - New context is created: duplicate of parent process
- *fork* returns 0 to the child and the process ID to the parent
 - Both processes execute at the point of the return from the *fork*



What happens in *fork*?

- Check for available resources
- Allocate a new PCB
- Assign a unique PID
- Check process limits for user
- Set child state to “created”
- Copy data from parent PCB slot to child
- Increment counts on current directory & open files
- Copy parent context in memory (or set *copy on write*)
- Set child state to “ready to run”
- Wait for the scheduler to run the process

Fork Example

```
#include <stdio.h>

main(int argc, char **argv) {
    int pid;

    switch (pid=fork()) {
        case 0:    printf("I'm the child\n");
                    break;
        default:
            printf("I'm the parent of %d\n", pid);
            break;
        case -1:
            perror("fork");
    }
}
```

Running other programs

execve: replace the current process image with a new one

- See also *execl*, *execle*, *execvp*, *execvP*
(these are just variation wrappers that take different parameters)

- New program inherits:

- Processes group ID
- Open files
- Access groups
- Working directory
- Root directory
- Resource usages & limits
- Timers
- File mode mask
- Signal mask

Exec Example

Execute the command: `ls -al /`

```
#include <unistd.h>

main(int argc, char **argv) {
    char *av[] = { "ls", "-al", "/", 0 };

    execvp("ls", av);
    perror("ls failed to run!");
    exit(1);
}
```

The *perror* and *exit* functions run ONLY if *execvp* failed – otherwise the new program overlays the current process

Fork & exec combined

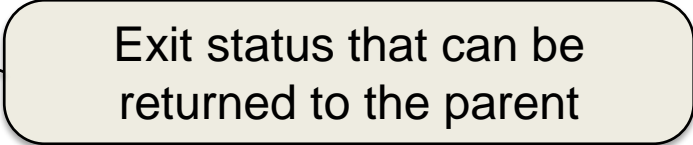
- UNIX runs new programs via *fork* followed by *exec*
 - Step 1. Clone
 - Step 2. Replace
- Windows approach
 - *CreateProcess* system call to create a new child process
 - Specify the executable file and parameters
 - Identify startup properties (windows size, input/output handles)
 - Specify directory, environment, and whether open files are inherited

Exiting a process

exit system call

```
#include <stdlib.h>

main(int argc, char **argv) {
    exit(0);
}
```



Exit status that can be returned to the parent

exit: what happens?

- Ignore all signals
- If the process is associated with a controlling terminal
 - Send a hang-up signal to all members of the process group
 - reset process group for all members to 0
- close all open files
- release current directory
- release current changed root, if any
- free memory associated with the process
- write an accounting record (if accounting)
- make the process state zombie
- assign the parent process ID of any children to be 1 (init)
- send a “death of child” signal to parent process (SIGCHLD)
- context switch (we have to!)

Wait for a child process to die

wait system call

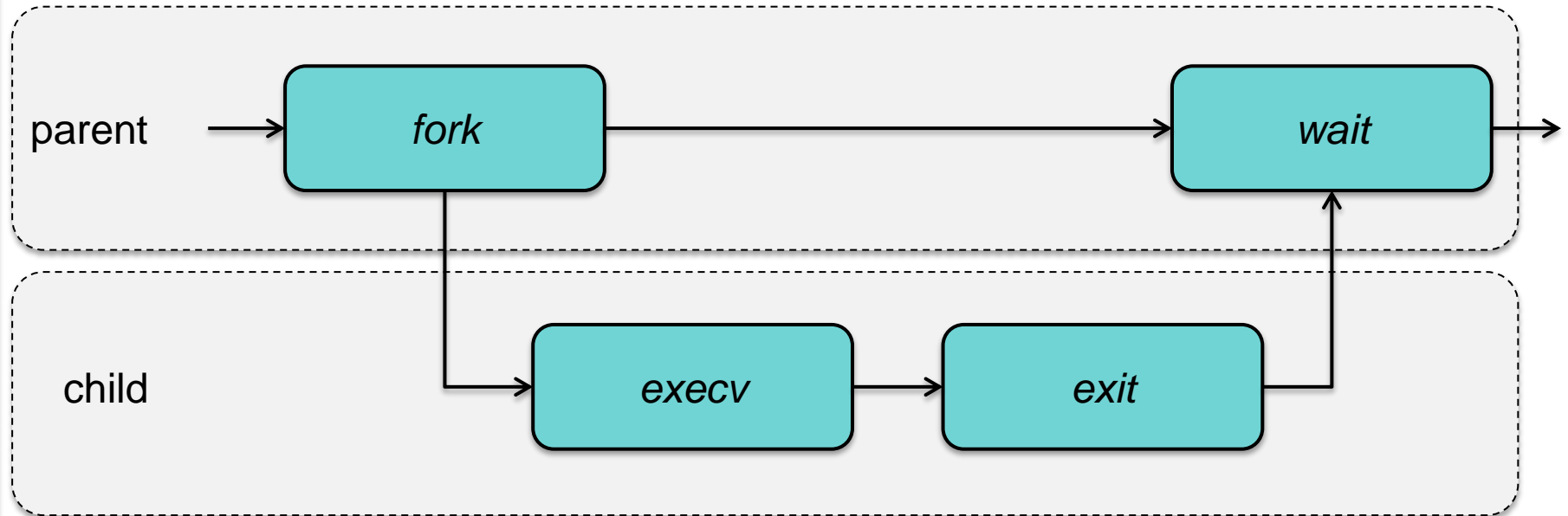
- Suspend execution until a child process exits
- *wait* returns the exit status of that child.

```
int pid, my_pid, status;

switch (my_pid=fork()) {
case 0:          /* do child stuff */ break;
case -1:         /* do error stuff */ break;

default:         /* wait for child to exit */
    pid=wait(&status);
    if (pid != -1)
        printf("got exit of %d\n", WEXITSTATUS(status));
        break;
}
```

Parent & child processes



Signals

- Inform processes of asynchronous events
 - Processes may specify signal handlers
- Processes can poke each other (if they are owned by the same user)
- Sending a signal:
 - *kill* (*int pid, int signal_number*)
- Detecting a signal:
 - *signal* (*signal_number, function*)

The End