

Operating Systems Design

4. Processes

Paul Krzyzanowski
pxk@cs.rutgers.edu

Key concepts from last week

Boot Loader

- Multi-stage boot loader
- Traditional Intel PC architecture
 - BIOS
 - Master Boot Record
 - Volume Boot Record
 - OS Loader
- Newer PC architecture (2005+)
 - UEFI – knows how to read one or more file systems
 - Loads OS loader from a boot partition
- Embedded systems
 - Boot firmware on chip

Operating System vs. Kernel

- Kernel
 - “nucleus” of the OS; main component
 - Provides abstraction layer to underlying hardware
 - 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

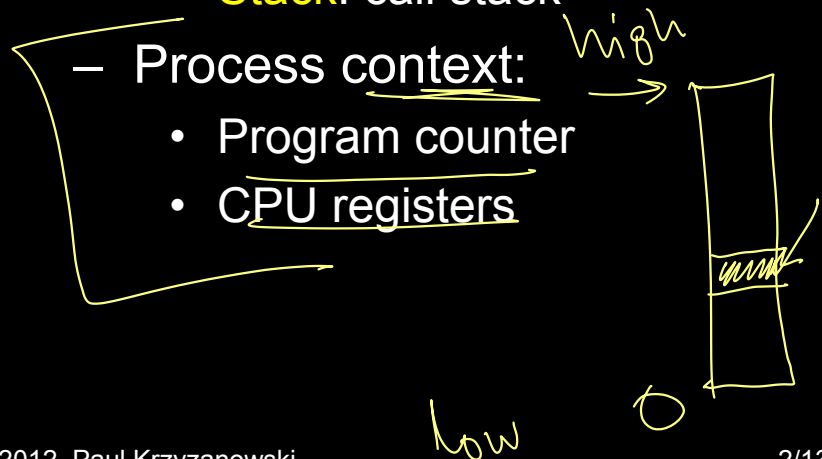
Timer interrupts

- 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 (Advanced Programmable Interrupt Controller; 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: aperiodic – tickless kernel
 - PIT not used for periodic interrupts; just APIC timer interrupts

Processes

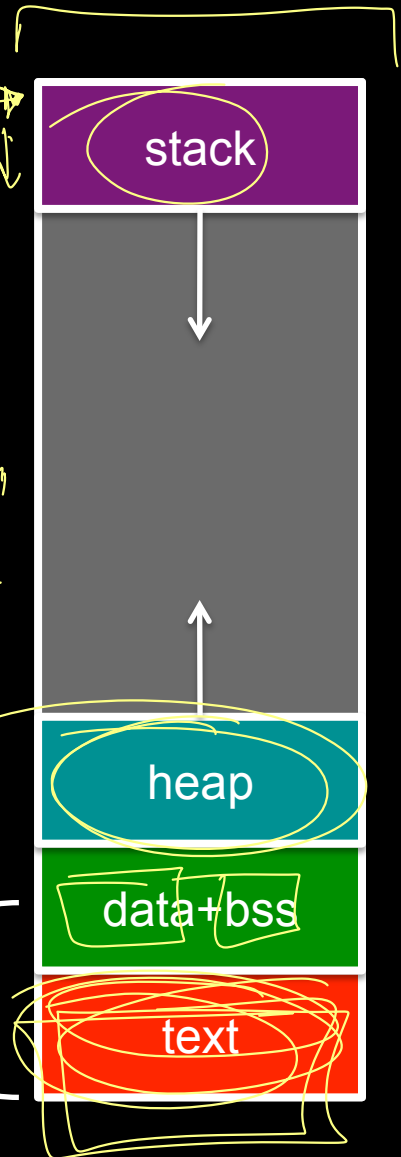
Process

- **Program**: code & static data stored in a file
- **Process**: a program's execution context
 - Each process has its own address space
 - Memory map
 - **Text**: compiled program
 - **Data**: initialized static data
 - **BSS**: uninitialized static data
 - **Heap**: dynamically allocated memory
 - **Stack**: call stack
 - Process context:
 - Program counter
 - CPU registers

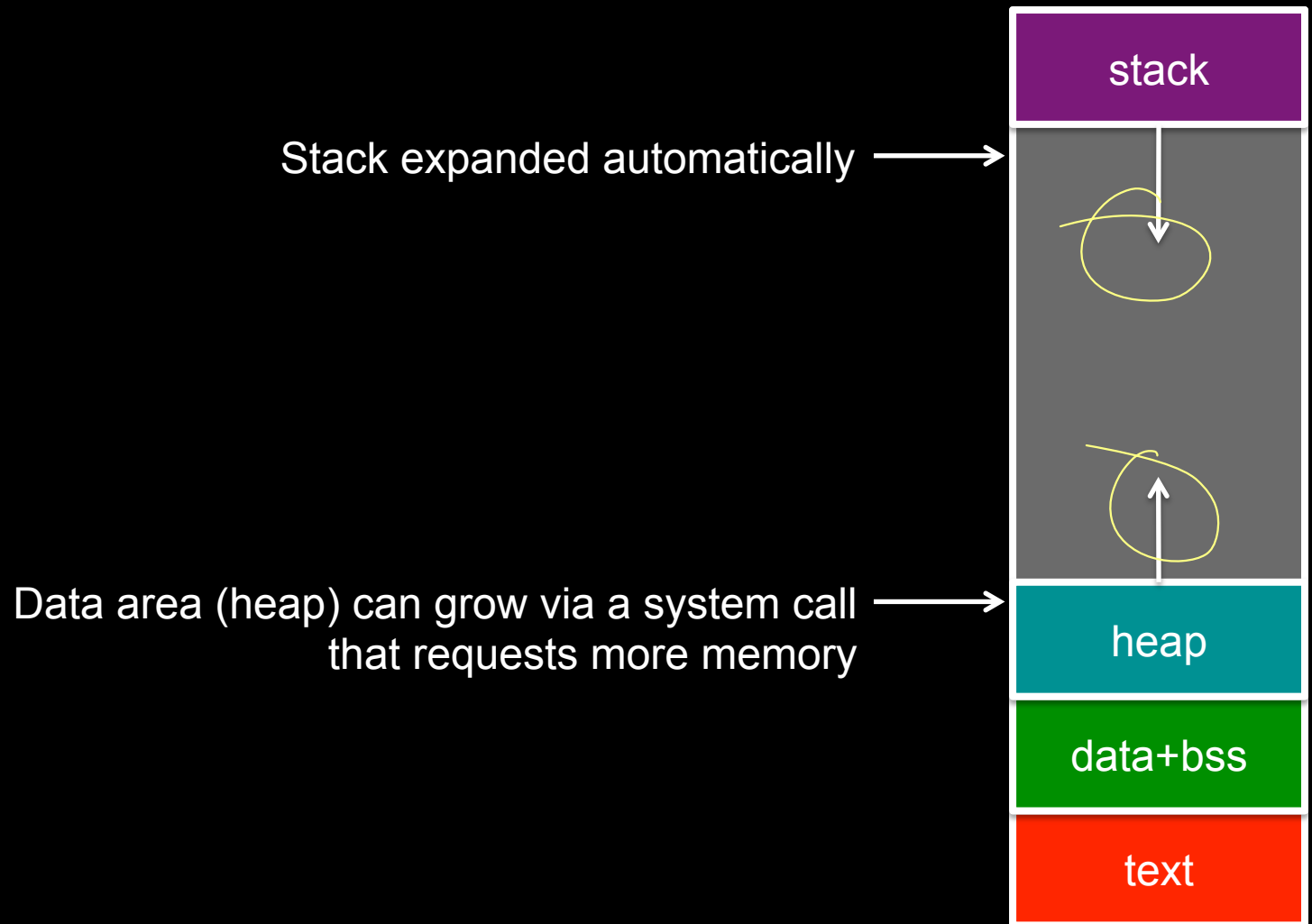


Text (code) & initialized data come from the stored program

"stack grows downwards"



Growing memory



Contexts

- Entering the kernel

- ① – Hardware interrupts

- Asynchronous events (I/O, clock, etc.)

- Do not relate to the context of the current process
 - Because they are asynchronous, *any* process might be running when they occur

- ② – Software traps

- Are related to the context of the current process [process context]
 - Examples: illegal memory access, divide by zero, illegal instruction

- ③ – Software initiated traps

- System call from the current process [process context]

- *The current 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, ...

System calls

- **Entry: Trap to system call handler**

- Save state
- Verify parameters are in a valid address
- Copy them to kernel address space
- Call the function that implements the system call
 - If the function has to (cannot be satisfied immediately) then
 - Context switch to let another ready process run
 - Put our process on a blocked list

- **Return from system call or interrupt**

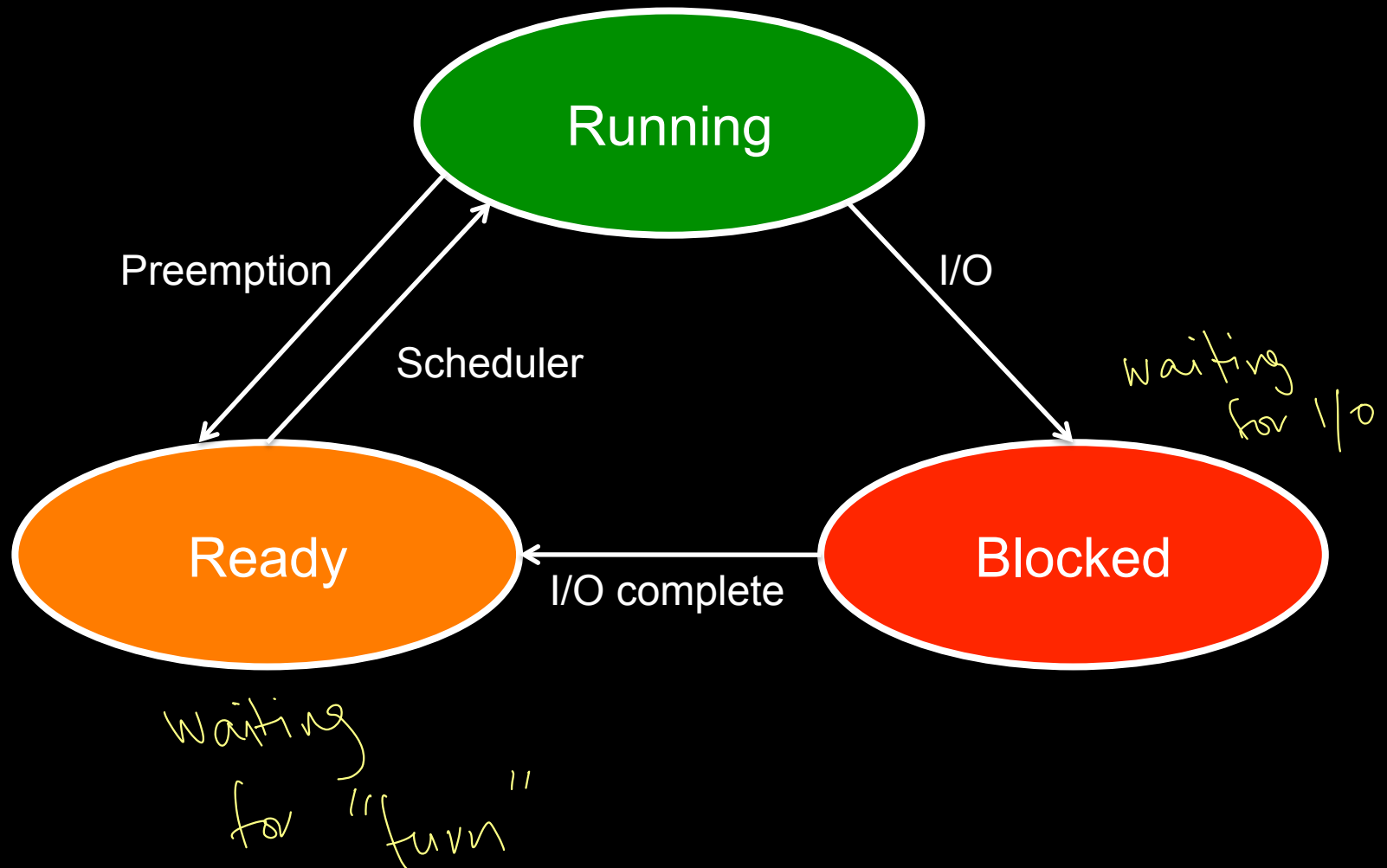
- Check for signals to the process
 - Call the appropriate handler if signal is not ignored
- Check if another process should run
 - Context switch to let the other process run
 - Put our process on a *ready* list
- Calculate time spent in the call for profiling/accounting
- Restore user process state
- Return from interrupt

“housekeeping”

Processes in a Multitasking Environment

- Multiple concurrent processes
 - Each has a unique identifier: Process ID (PID)
- Asynchronous events (interrupts) may occur 1/0
- Processes may request operations that take a long time
- Goal: have some process running at all times
- Context saving/switching
 - Processes may be suspended and resumed
 - Need to save all state about a process so we can restore it

Process states

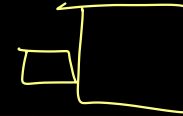


Keeping track of processes

- 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)

Shunt

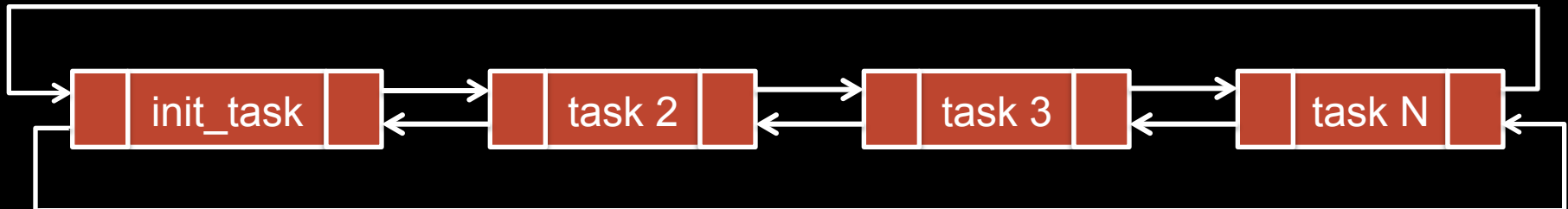
Processes in Linux



- 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 list head in `linux/list.h`

linked list
of "task struct"

```
struct task_struct init_task; /* static definition */
```



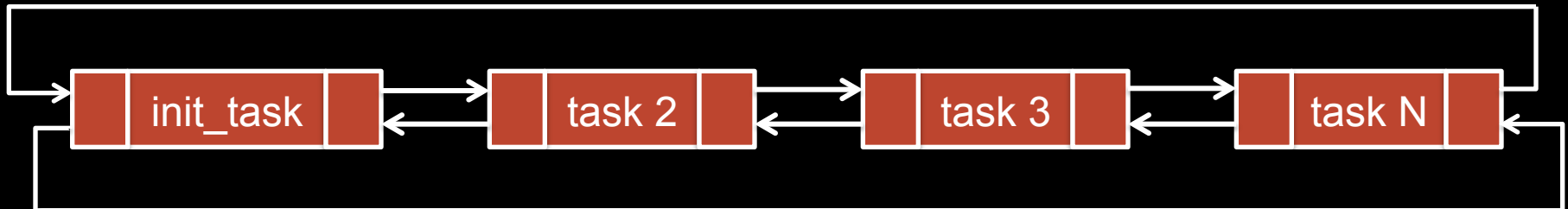
Processes in Linux

- Iterating through processes


```
for (p = &init_task; ((p = next_task(p)) != &init_task; ) {  
    /* whatever */  
}
```

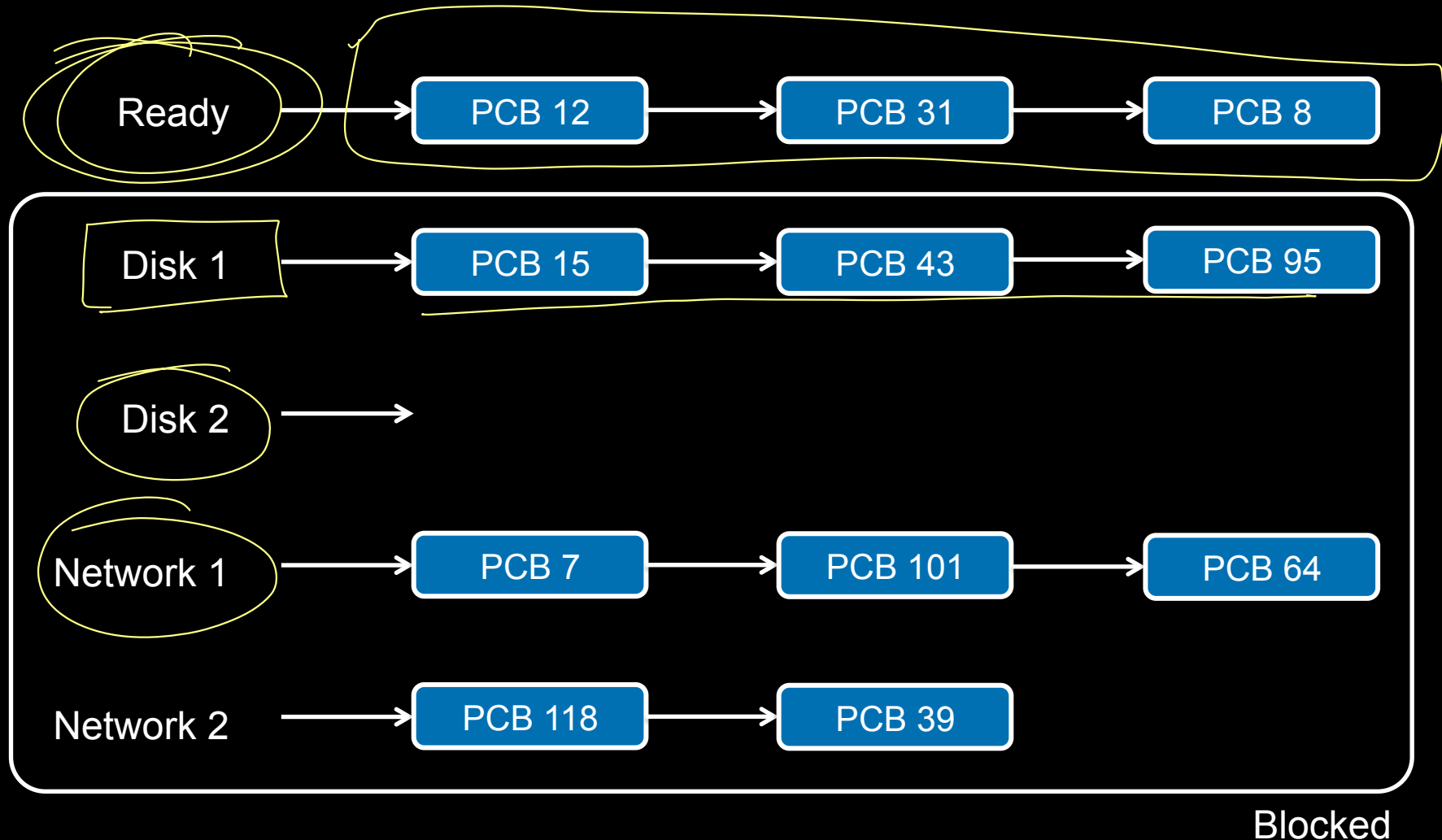
- The current process on the current CPU is obtained from the ~~macro~~ **current**

```
current->state = TASK_STOPPED;
```

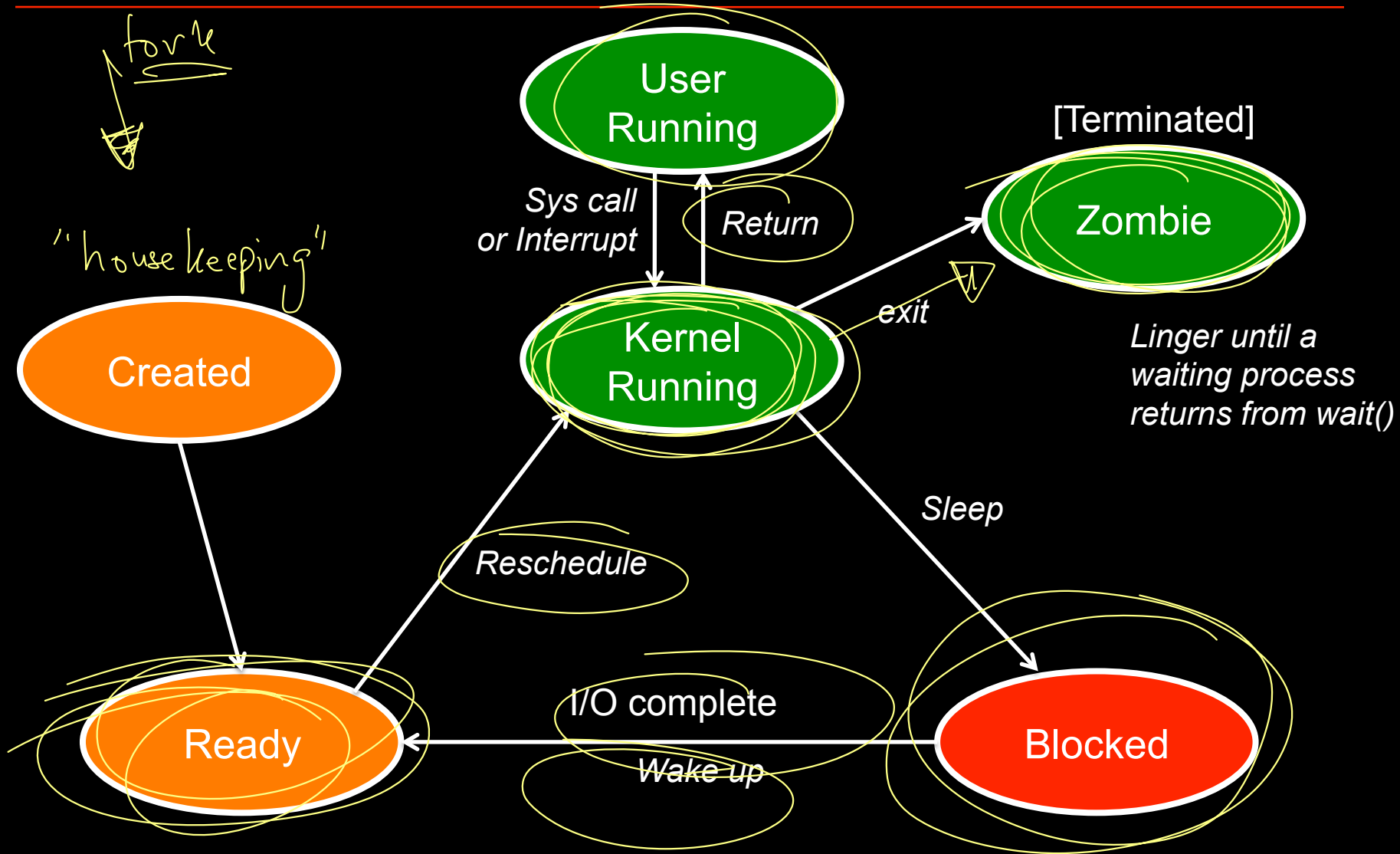


Processes on Ready & Blocked Queues

Running 



Process States: a bit more detail



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

What happens?

- 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()

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");
    }
}
```

A red arrow originates from the `fork()` function call in the `switch` statement and points to the `case 0:` label, indicating that the child process continues execution from this point.

==

int pid;

0 = fork

pid = fork();

20 = fork

if (pid == 0) {

printf("child");

} else {

printf("parent");

}

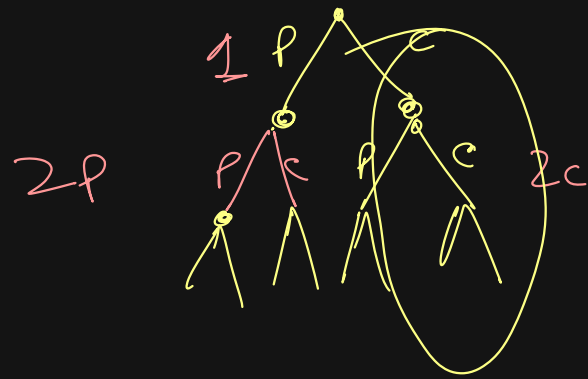
exit(0) →

fork(); 1

fork(); 2

A handwritten diagram on a black background. It features a large yellow-outlined box. Inside the box, there are two yellow arrows pointing from left to right towards the text 'fork()'. The text 'fork()' is written in yellow, followed by a yellow semicolon ';'. To the right of the box, there is a large red number '3'.

```
sleep(4000000);
```



Running other programs

execve: replace the current process image with a new one

– See also *execl*, *execle*, *execlp*, *execvp*, *execvP*

- 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

```
#include <unistd.h>

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

    execvp("ls", av);
}
```

Fork & exec combined

- UNIX creates processes via *fork* followed by *exec*
- 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: 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

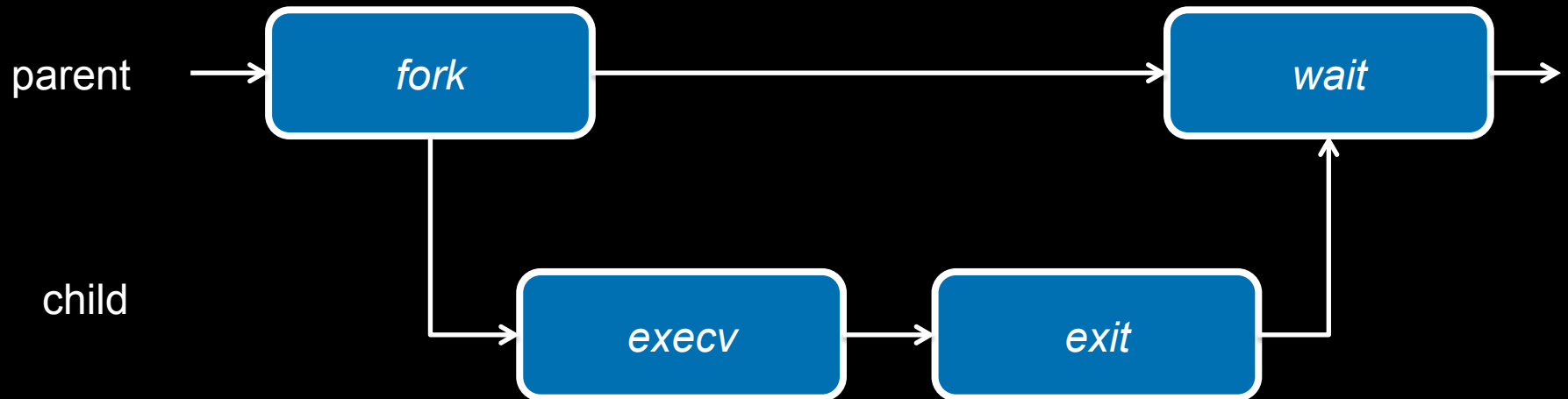
- Suspend execution until a child process exits
- *wait* returns the exit status of that child.

```
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 */
    while (pid=wait(&status))
        if (pid==my_pid)
            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