Processes

Dr. Yingwu Zhu

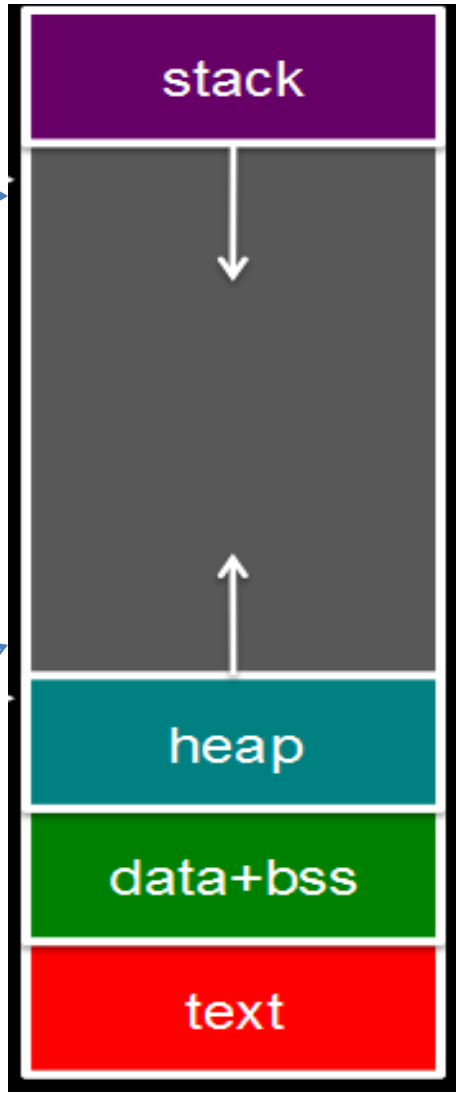
Process

- 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

Growing Memory

Stack expands automatically

Data area (heap) can grow via a system call that requests more memory - malloc() in c/c++



Contexts

- Entering the kernel (mode)
  - Hardware interrupts
    - Asynchronous events (I/O, clock, etc.)
    - Do not relate to the context of the current process
  - Software traps
    - Are related to the context of the current procesds
    - E.g., illegal memory access, divide by zero, illegal instruction
  - Software initiated traps
    - System call from the current process
  - *The current executing process's 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
  - 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 to run
      - Put our process in a blocked list

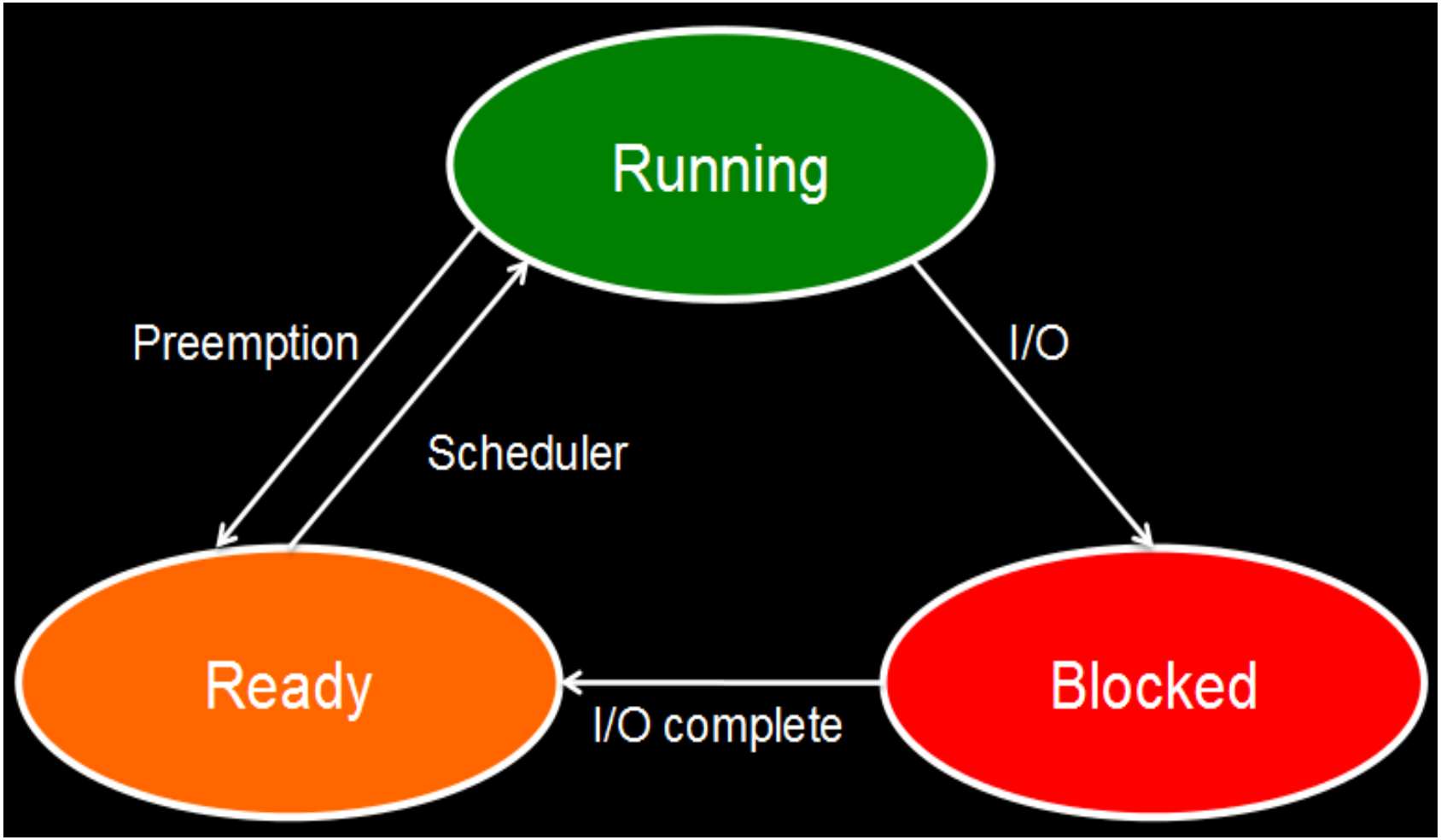
System Calls (cont)

- Return from system call or interrupt
  - Check for signals to the process
  - 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

Processes in a Multitasking Environment

- Multiple concurrent processes
  - Each has a unique identifier: PID
- Asynchronous events (interrupts) may occur
- Processes many request operations that take a long time
- Goals: 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
- PCB contains
  - PID
  - Machine state (registers, PC, 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)

PCB and Hardware State

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - The hardware registers contain the current values
- When the OS stops running a process, it saves the current values of the registers into the process' PCB
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process' PCB
- The process of changing the CPU hardware state from one process to another is called a context switch

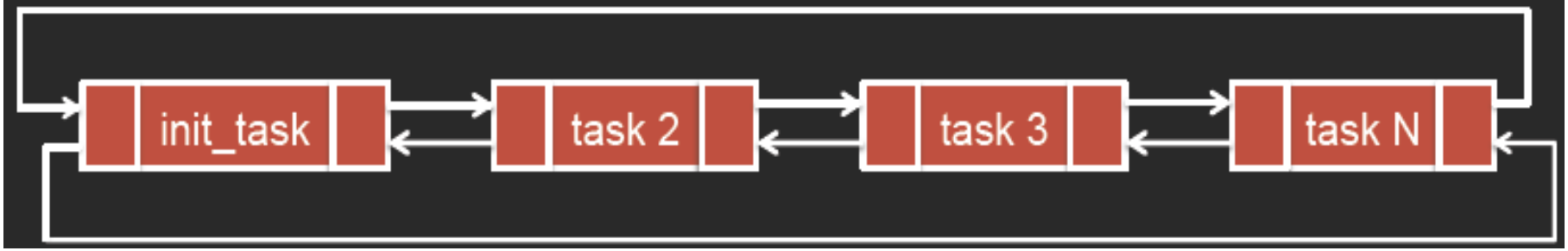
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does NO useful work while switching
- Time dependent on hardware support
  - Depends on
    - Memory speed, #-of-registers to copy, special instructions (single instruction to load/save all registers)
    - A few milliseconds. This can happen 100 or 1000 times a second!

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`

struct task_struct init_task; /* static definition */



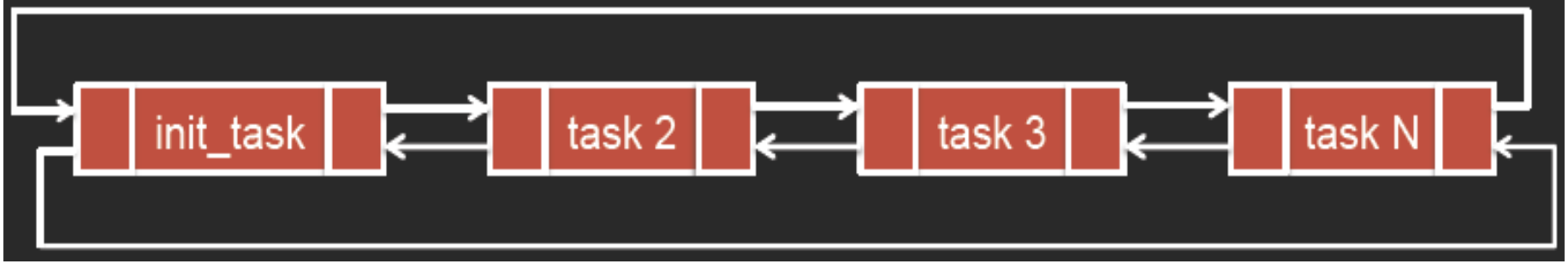
Processes in Linux

- Iterating through processes

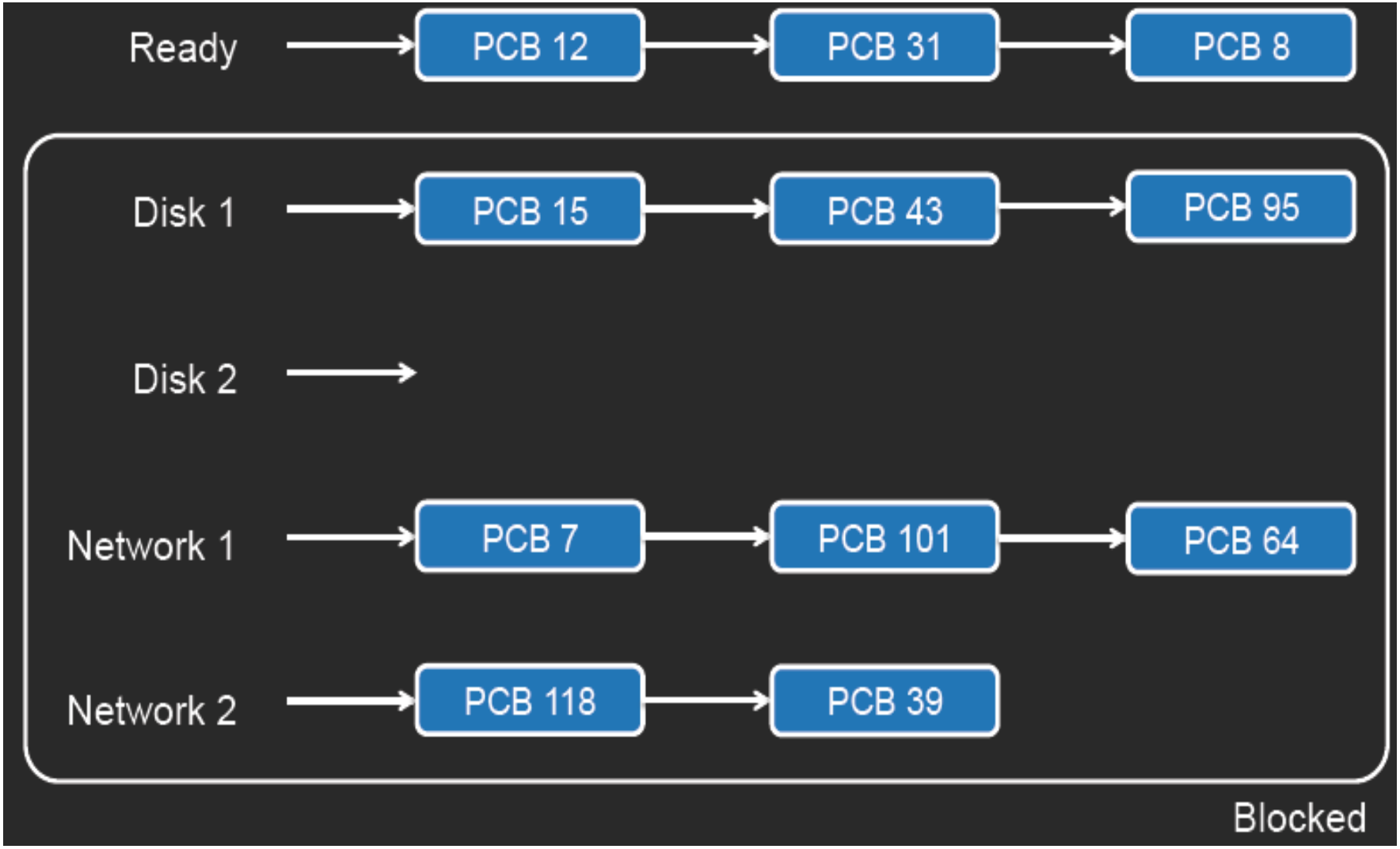
```
for (p = &init_task; ((p = next_task(p)) != &init_task; ) {
    /* whatever */
}
```

- The current process on the current CPU is obtained from the macro `current`

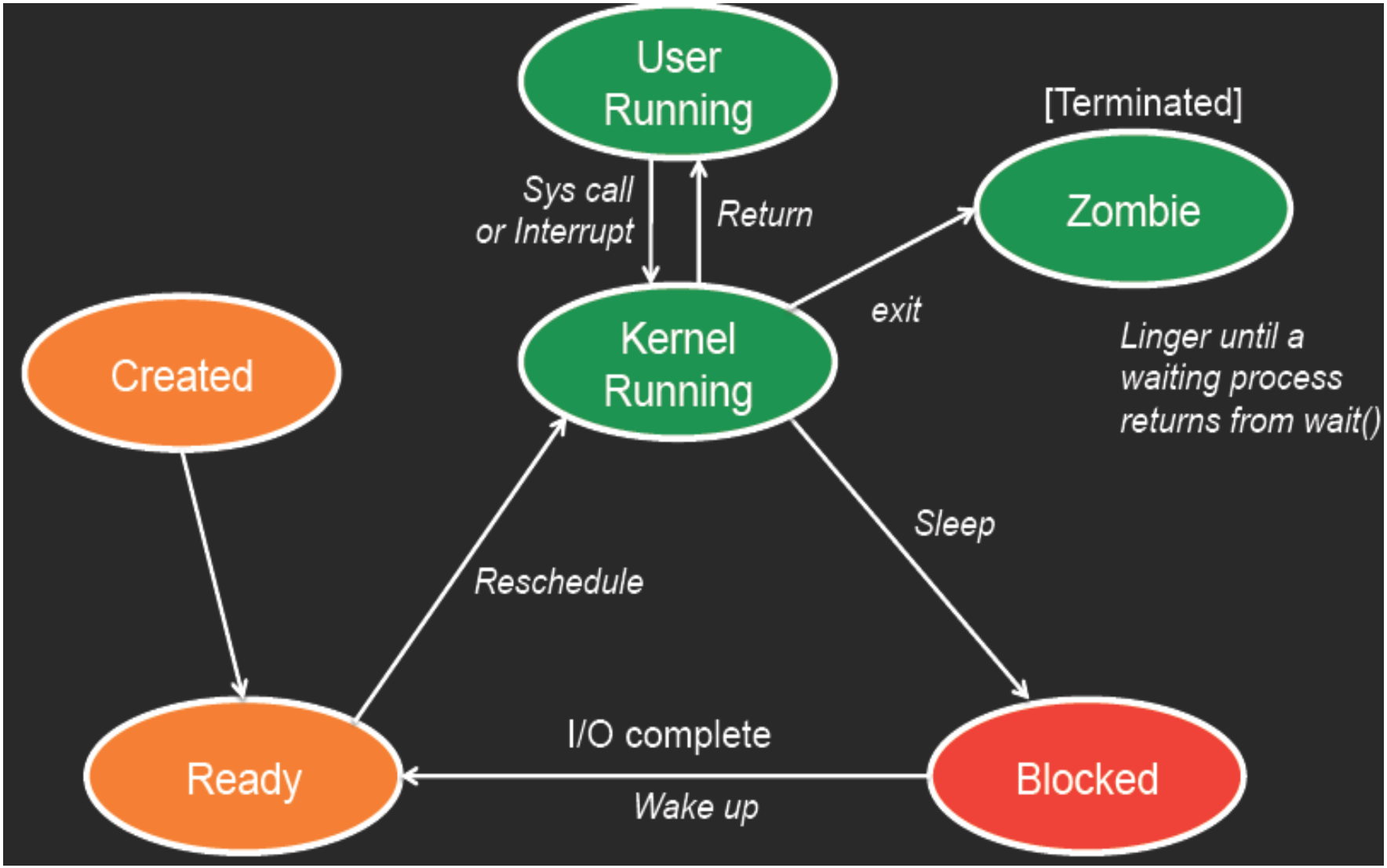
```
current->state = TASK_STOPPED;
```



Processes on Ready & Blocked Queues



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 process & 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 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 (*execl, execle, execlp, execvp*…)
- New process inherits
  - Process 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);
}
```

Exit a process

exit system call

```
#include <stdlib.h>

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

What happens for exit?

- Ignore all signals
- 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 PID of any children to be 1 (init)
- Send a “death of child” signal to parent process (SIGCHLD)
- Context switch

Wait for child process to die

wait system call

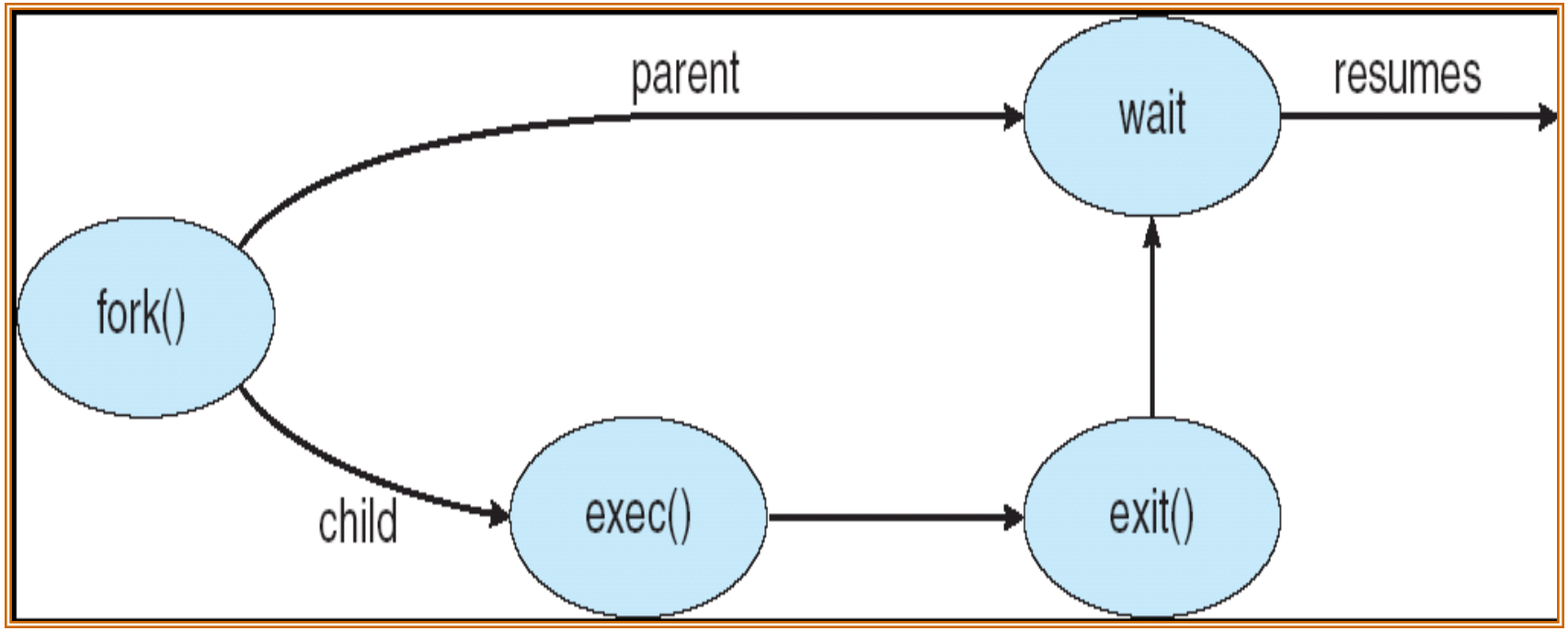
- Suspend execution until a child process exits
- *wait* returns the exit status of that child.

```
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 Process



What Kernel does for a process sleeping on a wait?

```
loop forever {
    if waiting process has a zombie child
        pick any zombie child
        add its CPU usage to the parent
        free child process control block
        return child ID and the exit code of the child
    if process has no children
        return error
    sleep at an interruptible priority
}
```

Signals

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

Signal Example

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

main(int argc, char **argv)
{
        if (fork())                     /* fork, assume it always works */
                for (;;)                /* parent prints a message forever */
                        printf("I'm the parent\n");
        else {  /* we're the child */
                sleep(3);               /* do nothing for three seconds */
                kill(getppid(), SIGKILL);               /* kill the parent */
        }
}
```

Detecting signals

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

main(int argc, char **argv)
{
        if (fork())     {       /* fork, assume it always works here */
                /* -------- parent -------- */
                void catchme();                 /* signal handling function */
                signal(SIGUSR1, catchme);       /* call catchme if we get SIGUSR1 */
                for (;;)        /* parent prints a message forever */
                        printf("I'm the parent\n");
        }
        else {  /* we're the child */
                sleep(3);       /* do nothing for 3 seconds */
                kill(getppid(), SIGUSR1);       /* send SIGUSR1 to the parent */
        }
}

void
catchme() {     /* signal handler */
        printf("got the signal!\n");
        exit(0);
}
```

How does kernel check for signals?

- Signals may occur asynchronously and often occur when the process is not running
- The kernel sets a bit in the PCB for that process upon a signal sent to it
- When the process is brought from ready to running, the kernel does…

How does kernel check for signals?

- When the process is brought from ready to running, the kernel does…

```
while ("received signal" field in PCB is not zero) {
    find a signal number set to the process
    if (signal == death of child)
        if ignoring death of child
            free PCBs of zombies for this parent
        else
            return the signal
    else if not ignoring the signal
        return the signal
    turn off the "received signal" bit in the process control block
}
return "no signal"
```

Acknowledgement

- Some slides are adapted from Dr. Paul Krzyzanowski