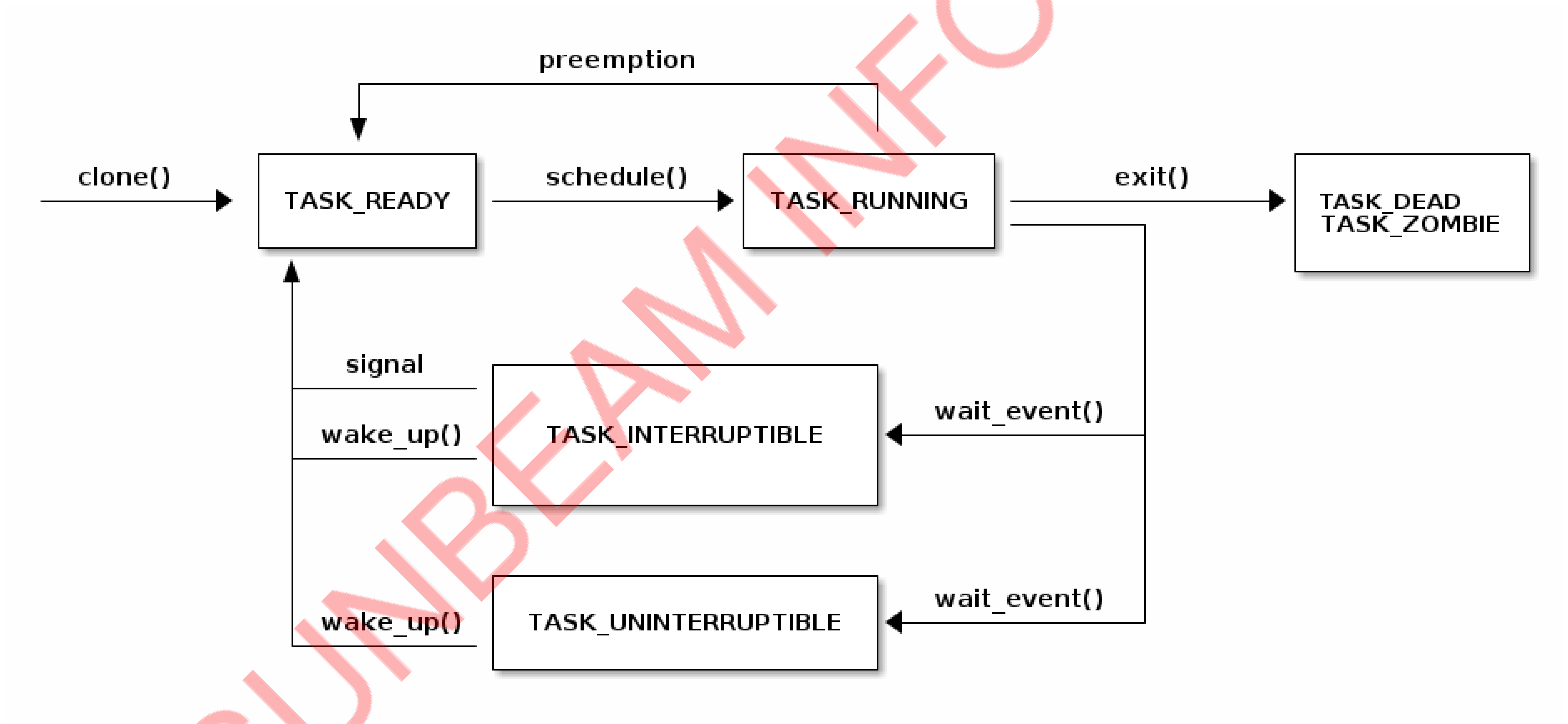
Embedded Operating Systems

Linux process

- https://linux-kernel-labs.github.io/refs/heads/master/lectures/processes.html

Linux Process Life Cycle



-
- TASK_READY (R)
- TASK_RUNNING (R)

- TASK_INTERRUPTIBLE (S)
- TASK_UNINTERRUPTIBLE (D)
- TASK_STOPPED (T)
- TASK_ZOMBIE (Z)
- TASK_DEAD (X)

Process Creation

- System Calls
  - Windows: CreateProcess()
  - Linux: clone(), fork(), vfork()
  - BSD UNIX: fork(), vfork()
  - UNIX: fork()

fork() syscall

- To execute certain task concurrently we can create a new process (using fork() on UNIX).
- terminal> man fork

```
#include <unistd.h>
pid_t fork(void);
```

  - fork() creates a new process by duplicating the calling process.
  - The new process is referred to as the child process. The calling process is referred to as the parent process.
  - The child process and the parent process run in separate memory spaces.
  - The child process is an exact duplicate of the parent process except for the following points:
    - The child has its own unique process ID
  - On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.
- Example:

```
#include <stdio.h>
#include <unistd.h>
int main() {
    int ret;
    printf("start!\n");
    ret = fork();
    printf("return value: %d\n", ret);
    printf("end!\n");
    return 0;
}
```

- fork() creates a new process by duplicating calling process.
- The new process is called as "child process", while calling process is called as "parent process".
- "child" process is exact duplicate of the "parent" process except few points pid, parent pid, etc.
- pid = fork();
    - On success, fork() returns pid of the child to the parent process and 0 to the child process.
    - On failure, fork() returns -1 to the parent.
- Even if child is copy of the parent process, after its creation it is independent of parent and both these processes will be scheduled sepeately by the scheduler.
- Based on CPU time given for each process, both processes will execute concurrently.

Questions on fork()

- Write such a program in which if and else both blocks are executing.

```
if(fork() == 0) {
    // child process
}
else {
    // parent process
}
```

- How fork() return two values i.e. in parent and in child?
  - fork() creates new process by duplicating calling process.
  - The child process PCB & kernel stack is also copied from parent process. So child process has copy of execution context of the parent.
  - Now fork() write 0 in execution context (r0 register) of child process and child's pid into execution context (r0 register) of parent process.
  - When each process is scheduled, the execution context will be restored (by dispatcher) and r0 is return value of the function.
- getpid() vs getppid()
  - pid1 = getpid(); // returns pid of the current process
  - pid2 = getppid(); // returns pid of the parent of the current process
- Why child execute statements only after the fork() and not before that?
  - fork() creates new process by duplicating calling process.
  - The child process PCB & kernel stack is also copied from parent process. So child process has copy of execution context of the parent. Note that the execution context was stored due to software interrupt.
  - Now child PC will be the same address as of parent process. The PC always stores address of next instruction to be executed. So when fork() was called, the PC contains address of next instruction (i.e. instruction after software interrupt).
  - When parent is scheduled it start executing from that address and when child scheduled, it continues execution from the same address (i.e. after software interrupt).
- When fork() will fail?
  - When no new PCB can be allocated, then fork() will fail.
  - Linux has max process limit for the system and the user. When try to create more processes, fork() fails.
  - terminal> cat /proc/sys/kernel/pid_max

Process hierarchy

- In UNIX/Linux each process have some parent process.
- The first process (process 1) is "init" process, which is created by a hardcoded "process 0".
- In newer Linux kernel versions "init" process is renamed as "systemd".
- terminal> ps -e -o pid,ppid,cmd

Orphan process

- If parent of any process is terminated, that child process is known as orphan process.
- The ownership of such orphan process will be taken by "init" process.

Zombie process

- If process is terminated before its parent process and parent process is not reading its exit status, then even if process's memory/resources is released, its PCB will be maintained. This state is known as "zombie state".
- To avoid zombie state parent process should read exit status of the child process. It can be done using wait() syscall.

wait() syscall

- ret = wait(&s);
  - arg1: out param to get exit code of the process.
  - returns: pid of the child process whose exit code is collected.
- wait() performs 3 steps:
  - Pause execution parent until child process is terminated.
  - read exit code from PCB of child process & return to parent process.
  - release PCB of the child process.
- The exit status returned by the wait() contains exit status, reason of termination and other details. Few macros are provided to access details from the exit code.
  - WEXITSTATUS()
  - WIFEXITED()

waitpid() syscall

- This extended version of wait() in Linux.
- ret = waitpid(child_pid, &s, flags);
  - arg1: pid of the child for which parent should wait.
    - -1 means any child.
  - arg2: out param to get exit code of the process.
  - arg3: extra flags to define behaviour of waitpid().

        - 0: same behaviour as of wait() i.e. wait for child termination.
        - WNOHANG: Do not wait for the child. Get exit code if child is already terminated, otherwise return error (-1).
    - returns: pid of the child process whose exit code is collected.
- waitpid(-1, &s, 0) is same as wait(&s).

Process Creation

- fork() creates a child process as duplicate of the calling process.
- One parent process can create multiple child processes by calling fork() multiple times.
- Parent should cleanup each child process it has created.

How fork() internally work?

```
// user space -- our application
int main() {
    int ret;
    ret = fork();
        // software interrupt
        // parent: ret=r0 (child pid), child: ret=r0 (0)
    printf("fork() returned: %d\n", ret);
        // fork() returned: child pid, child: fork() returned: 0
    return 0;
}
```

```
void swi_handler() {
    // save execution context on kernel stack (pc=line 166, r0-r14, psr)
    // get address of syscall impl from syscall table
    // invoke syscall impl
    // pid = cpu_scheduler()
    // restore execution context of selected process from its kernel stack
        // r0 (child pid or 0) and pc=line 166
}
```

```
// kernel space
int sys_fork() {
    // pseudo logic
    // allocate pcb for child
    // copy the pcb of parent into child
    // assign unique pid to child pcb
    // allocate memory for child process
    // copy the parent process into child process memory
    // return result;
        // parent kernel stack -- execution context -- r0 = child pid
        // child kernel stack -- execution context -- r0 = 0
}
```

sched_child_runs_first

- By default scheduler selects parent process to execute after fork() due to scheduler config.
  - terminal> cat /proc/sys/kernel/sched_child_runs_first
    - 0 -- false --> parent is selected first for execution
- This config can be modified so that child is selected first for the execution.
  - terminal> echo "1" | sudo tee /proc/sys/kernel/sched_child_runs_first
  - terminal> cat /proc/sys/kernel/sched_child_runs_first
    - 1 -- true --> child is selected first for execution
- This config works correctly in uni-processor environments.

exec() syscall

- exec() syscall "loads a new program" in the calling process's memory (address space) and replaces the older (calling) one.
- Example:

```
ret = fork();
if(ret == 0) {
    err = execl("/usr/bin/ls", "ls", ..., NULL);
    if(err < 0) {
        perror("execl() failed");
        _exit(1);
    }
}
else {
    wait(&staus);
}
```

- If exec() succeed, it does not return (rather new program is executed).
- There are multiple functions in the family of exec():
  - execl(), execlp(), execle(),
  - execv(), execvp(), execvpe()
  - execve() -- system call
- exec() family multiple functions have different syntaxes but same functionality.
  - l: variable argument List

    ```
    // ls -l -a /home
    err = execl("/usr/bin/ls", "ls", "-l", "-a", "/home", NULL);
        // execl("exeutable path", ..., NULL);
    ```

  - v: argument Vector (array)

    ```
    // ls -l -a /home
    char *args[] = { "ls", "-l", "-a", "/home", NULL };
    ```

```
err = execv("/usr/bin/ls", args);
    // execl("exeutable path", args_vector);
```

- p: find executable in the Path

```
env

echo $PATH
```

  - PATH contains set of directories separeted by ":".
  - When any program/command is executed on shell without giving it's full PATH, shell search it automatically in all directories given in PATH variable.
  - execlp() or execvp() automatically search executable in all directories in the PATH variable i.e. the first arg need not to be full path of executable.

```
err = execlp("ls", "ls", "-l", "-a", "/home", NULL);

// OR

err = execvp("ls", args);
```

- e: pass Environment variables to the child program.

  - Env variables contains important information about the system e.g. PATH, HOME, USER, SHELL, etc.

    ```
    env

    echo $USER
    ```

```
echo $SHELL
```

- To access Environment variables in a C program, use 3rd of main().

```
int main(int argc, char *argv[], char *envp[]) {
    int i;
    for(i=0; envp[i]!=NULL; i++)
        puts(envp[i]);
    return 0;
}
```

- execve(), execle() and execvpe() can pass Environment variables to the child program.

```
char *env_array[] = { "USER=test", "SHELL=/bin/csh", "HOME=/home/test" };

err = execle("/child.out", "child.out", ..., NULL, env_array);

// OR

err = execve("/child.out", args, env_array);
```

Process creation

fork() syscall

- allocate pcb for child.
- copy the pcb of parent into child.
- assign unique pid to child pcb.
- allocate memory for child process.

- copy the parent process into child process memory.
- return 0 to child and child pid to parent.

exec() syscall

- If new program section sizes are not matching with calling process sections, then it release sections of the calling process and reallocate them as per need of new program.
- Read sections from given executable file and copy them into sections of the calling process.
- Reset heap and stack sections of the calling process to begin execution of new program.
- Place command line args and env variables in the stack of the calling process.
- In PCB modify the execution context so that pc is set to startup (entry point) of the given program.

_exit() syscall

- Release all resources occupied by the process (i.e. memory in user space, page table memory/kernel stack in kernel space, close all open files, ...)
- Write exit status (arg1) into the process's PCB.
- Send SIGCHLD signal to its parent process.

Assignment

1. From one parent create 5 child processes. Each child should run for 5 seconds and print count along with its pid. Parent should wait for all child processes to complete and clean all of them. Hint: use loop to fork() multiple child processes.
2. From a parent process (A) create a child process (B). The child (B) in turn creates new child process (C) and it (C) in turn create new child (D). All these processes should run concurrently for 5 seconds and cleaned up properly upon termination.
3. Find max number of child processes that can be created on Linux using C program? Hint: use fork() in infinite loop (wisely).