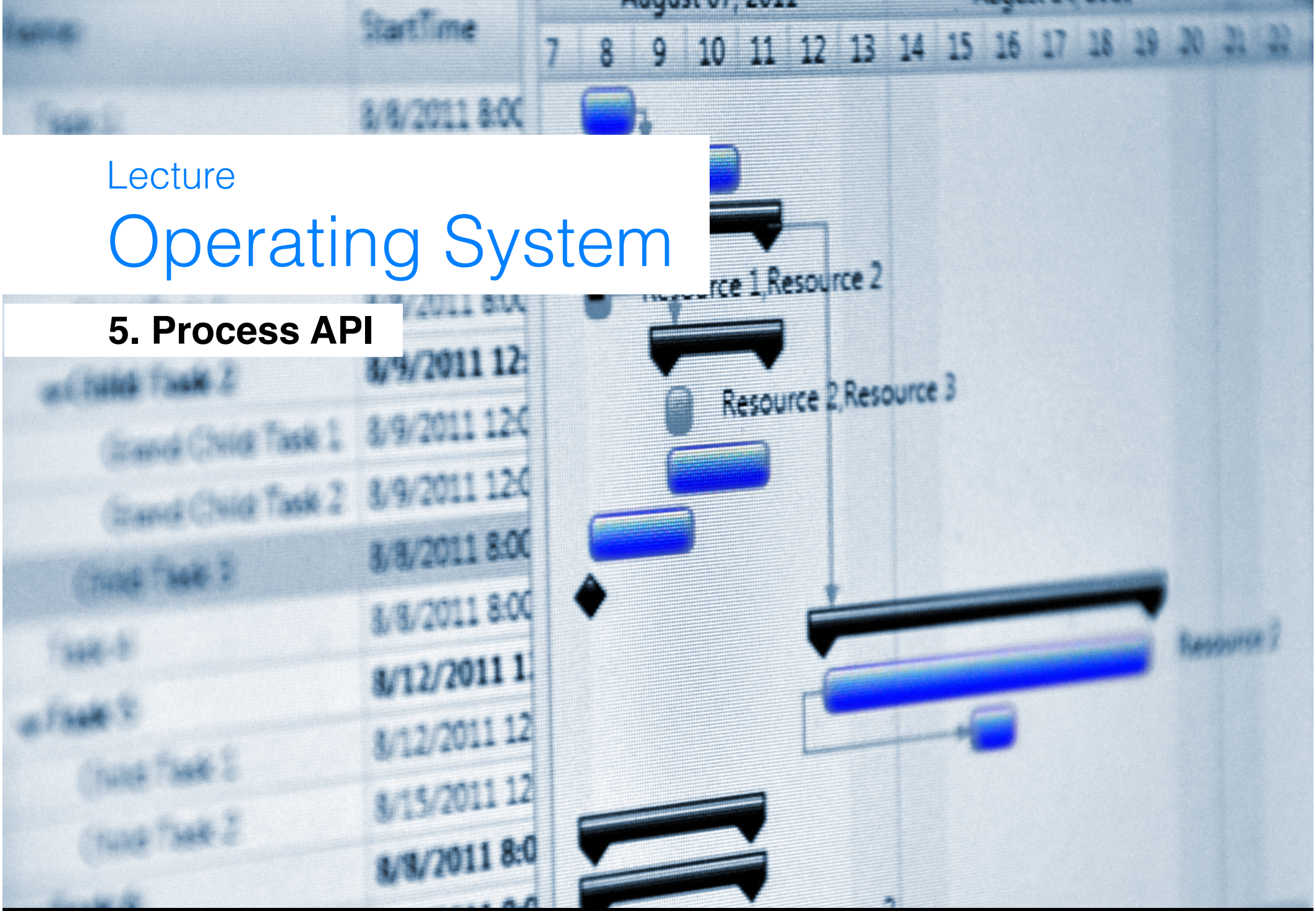
Lecture

Operating System

5. Process API



Professor Dr. Michael Mächtel

5. The Process API

1. **fork(), wait() and exec() System Call**
2. **fork() and kill() going bad**
3. **pipe() and dup2()**

5. The Process API

1. **fork(), wait() and exec() System Call**
2. **fork() and kill() going bad**
3. **pipe() and dup2()**

Two ways to create a process

- Build a new empty process from scratch
- Copy an existing process and change it appropriately

Option 1: New process from scratch

- **Steps**
  - Load specified code and data into memory;
    Create empty call stack
  - Create and initialize PCB (make look like context-switch)
  - Put process on ready list
- **Advantages**: No wasted work
- **Disadvantages**: Difficult to setup process correctly and to express all possible options
  - Process permissions, where to write I/O, environment variables
  - Example: WindowsNT has call with 10 arguments

Option 2: Clone existing process and change

- Example: Unix `fork()` and `exec()`
  - `fork()`: Clones calling process
  - `exec(char *file)`: Overlays file image on calling process
- `fork()`
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to child process?
- `exec(char *file)`
  - Replace current data and code segments with those in specified file
- **Advantages**: Flexible, clean, simple. Fork is fast with CoW!
- **Disadvantages**: Wasteful to perform copy and then overwrite of memory like in `exec()`.

The fork() System Call

- Create a new process
  - The newly-created process has its own copy of the **address space**, **registers**, and **PC**.
  - Syntax: `retval = fork()`
  - `fork()` creates identical copy of (parent-)process
  - Difference between child an parent: `retval` !
    - in parent-process: `PID` of child
    - in child-process: `0`

The fork() System Call: Example

fork.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

The fork() System Call: Executed

```
…
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
     } else {
        // parent goes down this path (original process)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
     }
     return 0;
}
```

Result (Not deterministic)

```
prompt> ./fork
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

```
prompt> ./fork
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

getpid()

- Why no error check of `getpid()` ?
  - From man page: “The `getpid()` and `getppid()` functions are always successful, and no return value is reserved to indicate”

The wait() System Call

This system call won't return until child has run and exited

wait.c

```
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        sleep(1);
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
           rc, wc, (int) getpid());
    }
    return 0;
}
```

Result (Deterministic)

```
prompt> ./wait
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

The exec() System Call Family

Run a program that is different from the calling program

exec.c

```c
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("exec.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
           rc, wc, (int) getpid());
    }
    return 0;
}
```

All of the above in C with redirection

redirect.c

```c
int main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./redirect.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
    // now exec "wc" ...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("redirect.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
    }
    return 0;
}
```

```
prompt> less redirect.output
      31     109     857 redirect.c
```

How are Unix shells implemented?

shell.c

```
While (1) {
    Char *cmd = getcmd();
    Int retval = fork();
    If (retval == 0) {
        // This is the child process
        // Setup the child's process environment here
        // E.g., where is standard I/O, how to handle signals?
        exec(cmd);
        // exec does not return if it succeeds
        printf("ERROR: Could not execute %s\n", cmd);
        exit(1);
    } else {
        // This is the parent process; Wait for child to finish
        int pid = retval;
        wait(pid);
    }
}
```

Terminate a process

- Programmer terminates:
  - in Unix by `exit()`
  - Windows: `ExitProcess()`
- OS terminates:
  - Bug: access violation, division by zero, ...
- Another process terminates in
  - UNIX by `kill()`, or from shell by **`kill -0 PID`**
  - Windows by `TerminateProcess()`

5. The Process API

1. fork(), wait() and exec() System Call
2. **fork() and kill() going bad**
3. pipe() and dup2()

fork and kill: how badly things can go

bad.c

```c
#include <signal.h>
#include <unistd.h>

int main(void) {
        pid_t child = fork();
        if (child) {  // in parent
                sleep(5);
                kill(child, SIGKILL);
        } else {  // in child
                for (;;);  // loop until killed
        }

        return 0;
}
```

man page of kill:

*If pid is -1, **sig shall be sent to all processes** (excluding an unspecified set of system processes) for which the process has permission to send that signal.*

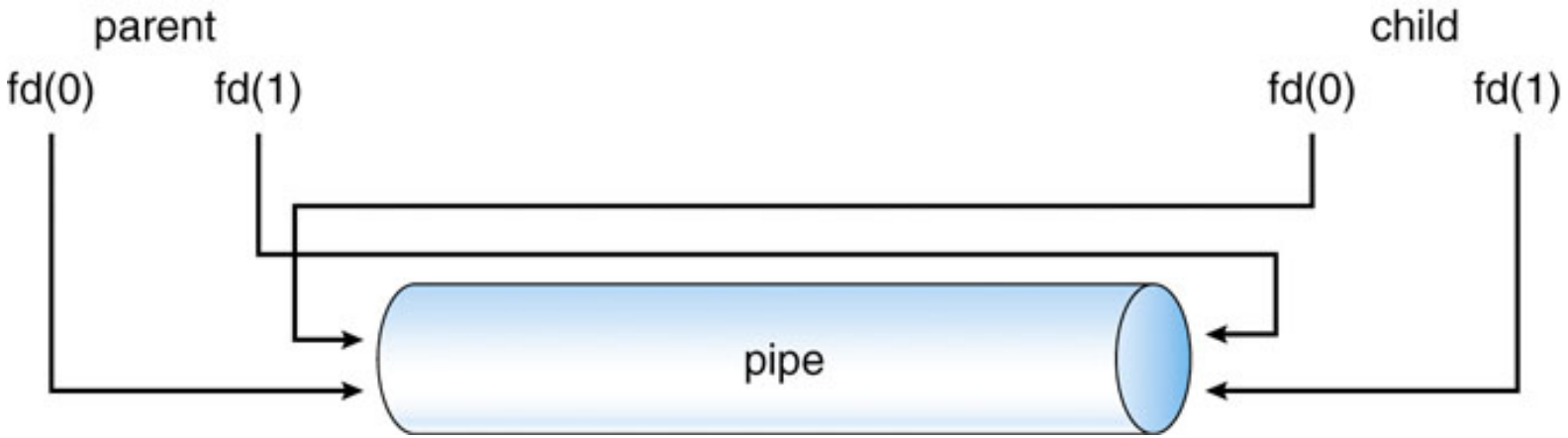
This program compiles with no errors or warnings, not even with `-Wall -Wextra -Werror`.

5. The Process API

1. fork(), wait() and exec() System Call
2. fork() and kill() going bad
3. **pipe() and dup2()**

The pipe() System Call

- `pipe(int fd[])`
  - Ordinary pipe (half duplex or duplex)
    - Typical: Producer -> Consumer



The pipe() System Call: Example

pipe.c

```
int main(void) {
  int fd[2], nbytes;
  pid_t childpid;
  char string[] = "Hello new Process!\n";
  char readbuffer[80];

  pipe(fd);

  if ((childpid = fork()) == -1) {
    perror("fork");
    exit(1);
  }
...
```

The pipe() System Call: Example

pipe.c

```
...

  if (childpid == 0) /* Child */
  {
    close(fd[1]); /* Close Output */
    nbytes = read(fd[0], readbuffer, sizeof(string) + 1);
    printf("Received string: %s", readbuffer);
    exit(0);
  } else /* Elternprozess */
  {
    close(fd[0]); /* Close Input */
    write(fd[1], string, strlen(string) + 1);
    wait(0); // wait for child, so process shutdowns
  }
  return (0);
}
```

```
> ./pipe
Received string: Hello new Process!
```

pipe() between processes

- Example: **ps ax | grep task1**
  - ps: `stdout` → to pipe
  - grep: `stdin` ← from pipe
- But how to connect `stdout` to `stdin` via pipe?
  - Systemcall: `dup2()`
    - `dup2(fd[1], STDOUT_FILENO)`
      - close `STDOUT` ('1') and reopen bound to write end of pipe
- Programming this Example means:
  - The shell has to wait for `grep task1` to finish
  - `grep task1` has to wait for `ps ax` to finish.

Thanks

Questions