Lecturer: Dr. Michael J. May
Kinneret College

Interrupts, Kernel Stack, Scheduler, SMT, `fork()`

28 November 2024
Lecture 4

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Dual mode

Base and bound

- Absolute
- With adder

uPC (user PC)

Syscall

Interrupt

Trap/Exception

Address space translation

Topics for Today

- Interrupt Handling and Kernel Stack
- Processes and Scheduler
- Digging Deeper: SMT
- `fork()`

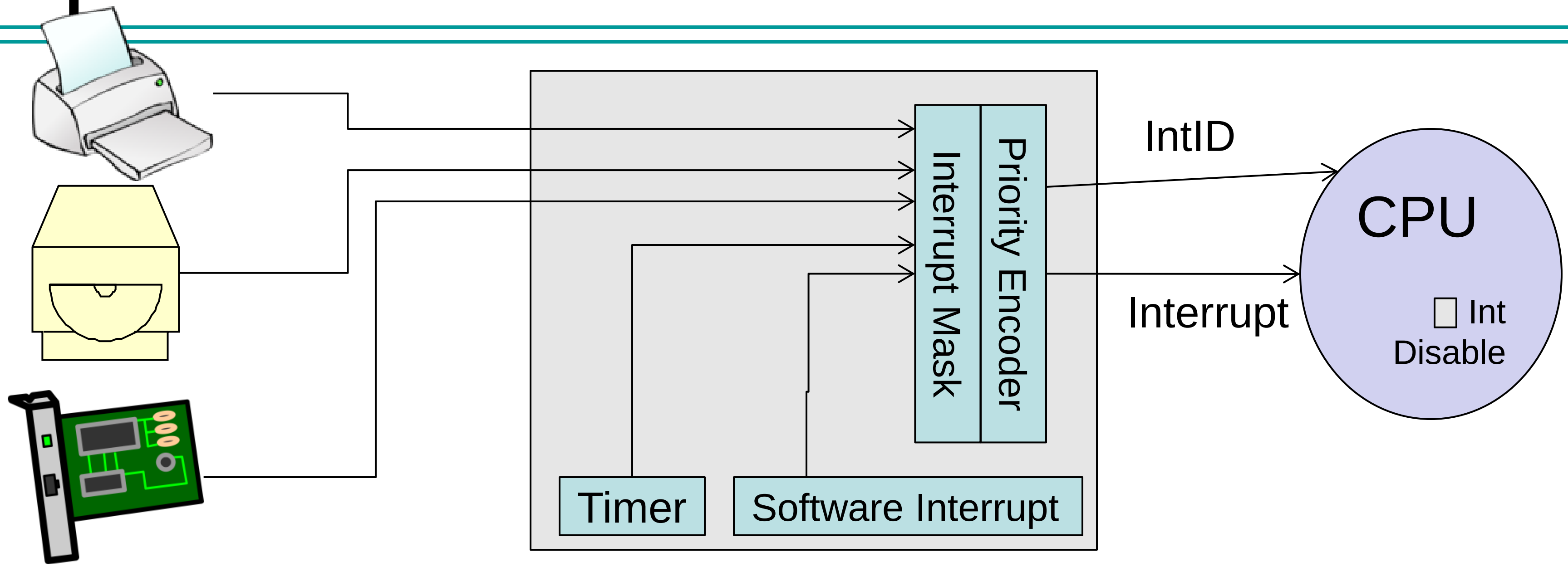
Concepts today

Hardware support: Interrupt Control

Interrupt Handler invoked with interrupts 'disabled'	OS kernel may enable/disable interrupts	Hardware may have multiple levels of interrupt

• Re-enabled upon completion
• Non-blocking (run to completion, no waits)
• Pack it up in a queue and pass off to an OS thread to do the hard work
  • Wake up an existing OS thread
	• On x86: CLI (disable interrupts), STI (enable)
	• Atomic section when selecting next process/thread to run
	• Atomic return from interrupt or syscall
		• Mask off (disable) certain interrupts, (ex. lower priority ones)
		• Certain non-maskable-interrupts (*nmi*)
		  • Ex. kernel segmentation fault

Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  – Mask enables/disables interrupts
  – Priority encoder picks highest enabled interrupt
  – Software Interrupt Set/Cleared by Software
  – Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- *Non-maskable interrupt line (NMI)* can't be disabled

How do we take interrupts safely?

Interrupt vector

- Limited number of entry points into kernel

Kernel interrupt stack

- Handler works regardless of state of user code

Interrupt masking

- Handler is non-blocking

Atomic transfer of control

- “Single instruction”-like to change the following:
  - Program counter
  - Stack pointer
  - Memory protection
  - Kernel/user mode

Transparent restartable execution

- User program does not know interrupt occurred

Interrupts on x86: Before

User-level Process

Code:

```
foo() {
    while (…) {
        x=x+1;
        y=y+1;
    }
}
```

Stack:

Registers

SS:ESP
CS:EIP
EFLAGS
Other registers: EAX, EBX

Kernel

Code:

```
Handler(){
    pusha
    …
}
```

Exception Stack:

Interrupts on x86: During

User-level Process

Code:

```
foo() {
    while (…) {
        x=x+1;
         y=y+1;
    }
}
```

Stack:

Registers

SS:ESP
CS:EIP
EFLAGS
Other registers: EAX, EBX

Kernel

Code:

```
Handler(){
    pusha
    …
}
```

Exception Stack:

SS
ESP
EFLAGS
CS
EIP
Error

Kernel System Call Handler

Locate arguments

- In registers or on user(!) stack

Copy arguments

- From user memory into kernel memory
- Protect kernel from malicious code evading checks

Validate arguments

- Protect kernel from errors in user code

Do something!

Copy back

- Copy results back into user memory

So Far

- Interrupt Handling and Kernel Stack
- Processes and Scheduler
- Digging Deeper: SMT
- `fork()`

Running Many Programs?

- We have the mechanism to
  - Switch between user processes and the kernel
  - Switch between user processes
  - Protect OS from user processes and processes from each other

Questions:

- How do we decide which user process to run?
- How does the OS record and manage user processes?
- How do we serialize the process and set it aside?
- How do we build a stack and heap for the kernel?
- How do we avoid wasting memory?

Process Control Block

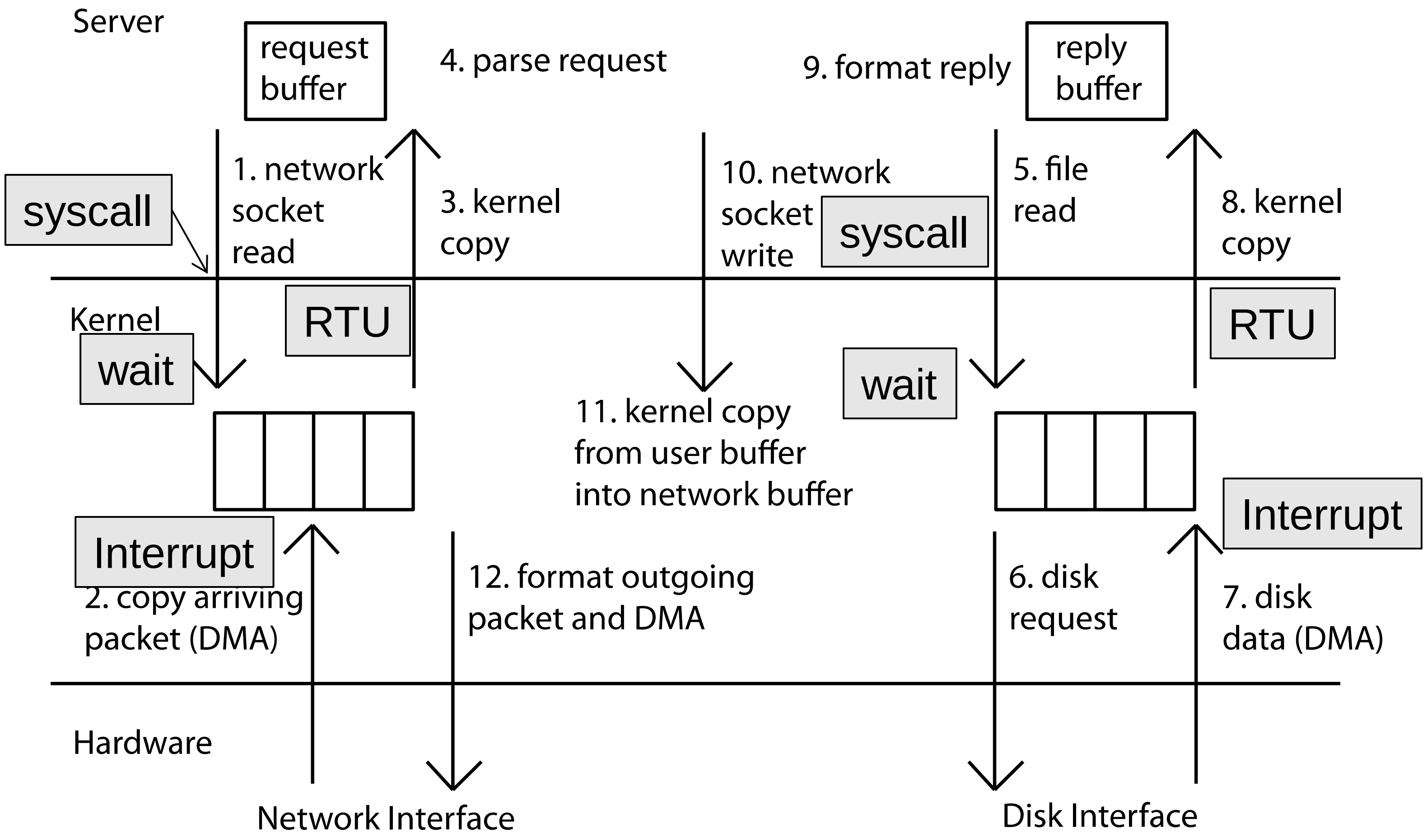
- Kernel represents each process as a Process Control Block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not running)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …

- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

Scheduler

```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

Putting it together: Web Server



So Far

- Interrupt Handling and Kernel Stack
- Processes and Scheduler
- Digging Deeper: SMT
- `fork()`

Digging Deeper:

SMT

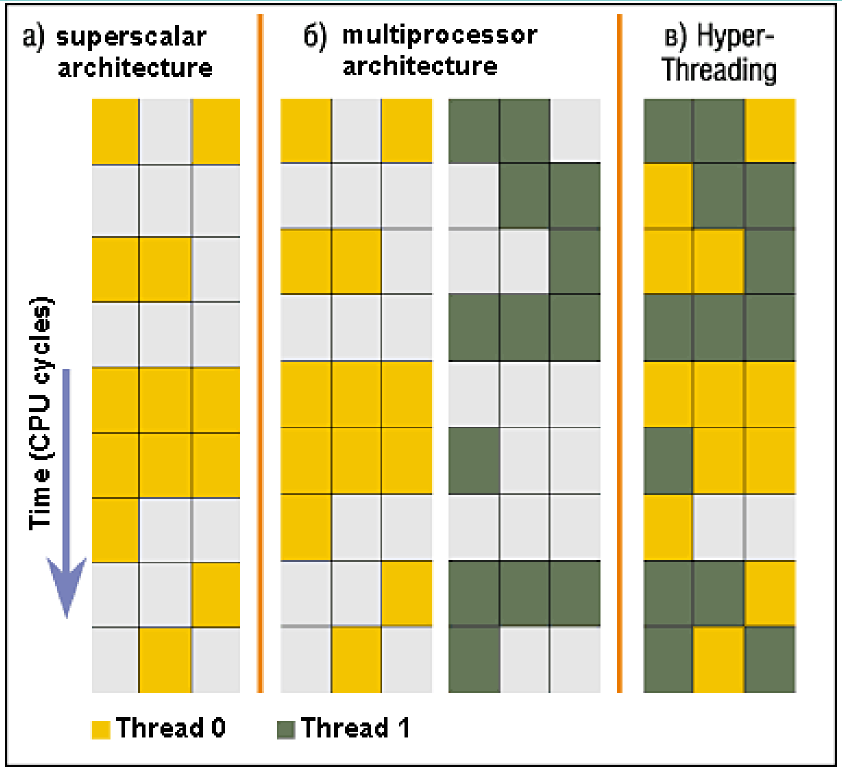
Simultaneous MultiThreading/Hyperthreading

Hardware technique

- Superscalar processors can execute multiple instructions that are independent.
- Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.

Can schedule each thread as if were separate CPU

- But, sub-linear speedup!



Colored blocks show instructions executed

Original technique called “Simultaneous Multithreading”

- http://www.cs.washington.edu/research/smt/index.html
- SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5

```
Item    Value
OS Name Microsoft Windows 10 Pro
Version 10.0.17134 Build 17134
Other OS Description    Not Available
OS Manufacturer Microsoft Corporation
System Name
System Manufacturer Dell Inc.
System Model    Latitude 5590
System Type x64-based PC
System SKU  0817
Processor   Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz, 2112 Mhz, 4 Core(s), 8 Logical Processor(s)
BIOS Version/Date   Dell Inc. 1.3.2, 6/8/2018
```

So Far

- Interrupt Handling and Kernel Stack
- Processes and Scheduler
- Digging Deeper: SMT
- `fork()`

Can a process create a process? How?

Option 1: Build a new one Windows

1. Allocate resources for new process – memory, virtual CPU, etc.
2. Assign task – an executable file
3. Assign ownership – user, parent process
4. Assign priority
5. Load task into an image
6. Mark it ready to run
7. Add it to ready queue

Who does steps 1-4?

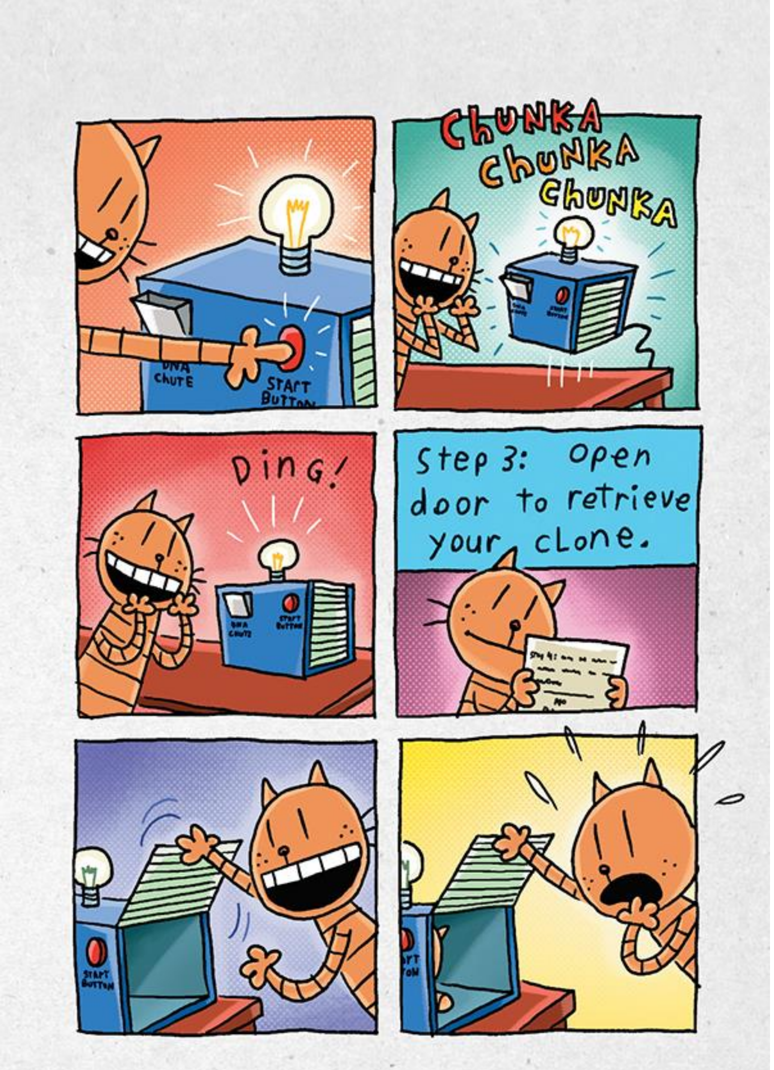
Option 2: Copy existing one

1. Copy all attributes of existing process
2. Change parts necessary in new process – task, memory, virtual CPU, priority, ownership
3. Make modified process child of original process
4. Run modified process

Who does steps 1-3?

https://openclipart.org/detail/284231/chicken-with-eggs

Cloning



[© 2024 Dav Pilkey]()

The UNIX/Linux way

Process ID

- Unique identity of process is the "process ID" (or pid).
- `fork()` creates a *copy* of current process with a new pid

State of original process duplicated in both Parent and Child!

- Memory, File Descriptors (next topic), etc

Return value from `fork()`: integer

- When > 0
  - Running in (original) Parent process
  - Return value is pid of new child
- When = 0
  - Running in new Child process
- When < 0
  - Error! Must handle somehow, running in original process

fork1.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  size_t readlen, writelen, slen;
  pid_t cpid, mypid;
  pid_t pid = getpid();         /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                         /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {   /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

How to tell them apart?

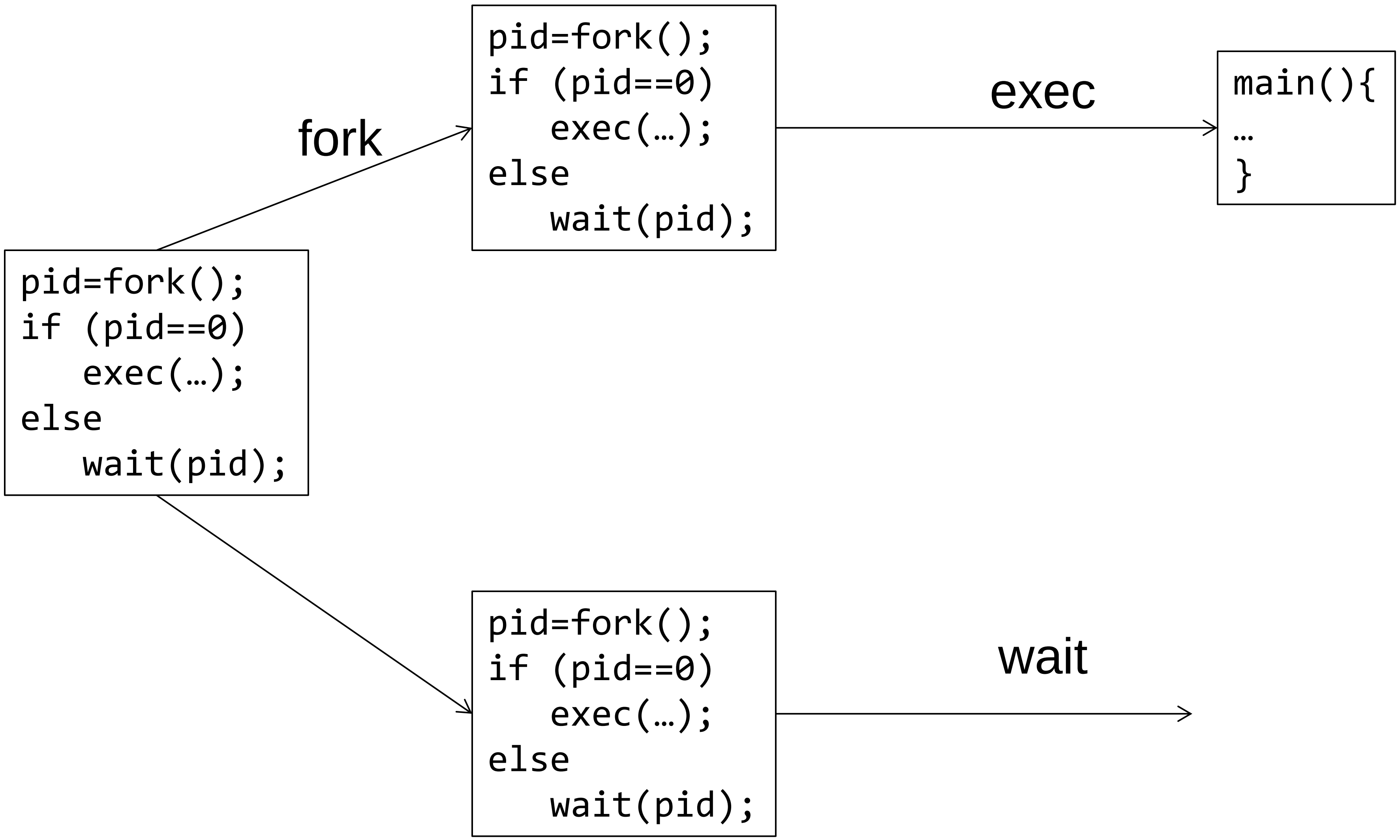
- http://www.gocomics.com/calvinandhobbes/1990/01/09
- http://www.gocomics.com/calvinandhobbes/1990/01/11

https://meaww.com/orphan-black-the-next-chapter-tatiana-maslany-teases-new-clones-fate-of-old-characters-10-years

fork2.c

```
int status;
…
cpid = fork();
if (cpid > 0) {                    /* Parent Process */
      mypid = getpid();
      printf("[%d] parent of [%d]\n", mypid, cpid);
      tcpid = wait(&status);
      printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
}  else if (cpid == 0) {           /* Child Process */
      mypid = getpid();
      printf("[%d] child\n", mypid);
}
…
```

Fork and Wait



Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- Example: to compile a C program

```
cc –c sourcefile1.c
cc –c sourcefile2.c
ln –o program sourcefile1.o sourcefile2.o
./program
```

process-race.c

```
int i;
cpid = fork();
if (cpid > 0 ) {
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
   for (i = 0; i< 100; i++) {
      printf("[%d] parent: %d\n", mypid, i);
      // sleep(1);
   }
}
else if (cpid == 0) {
   mypid = getpid();
   printf("[%d] child\n", mypid);
   for (i = 0; i > -100; i--)  {
      printf("[%d] child: %d\n", mypid, i);
      // sleep(1);
   }
}
```

What does this program print?
Does it change if you add the sleep?

Conclusion

- Interrupt Handling and Kernel Stack
- Processes and Scheduler
- Digging Deeper: SMT
- `fork()`