

Processes

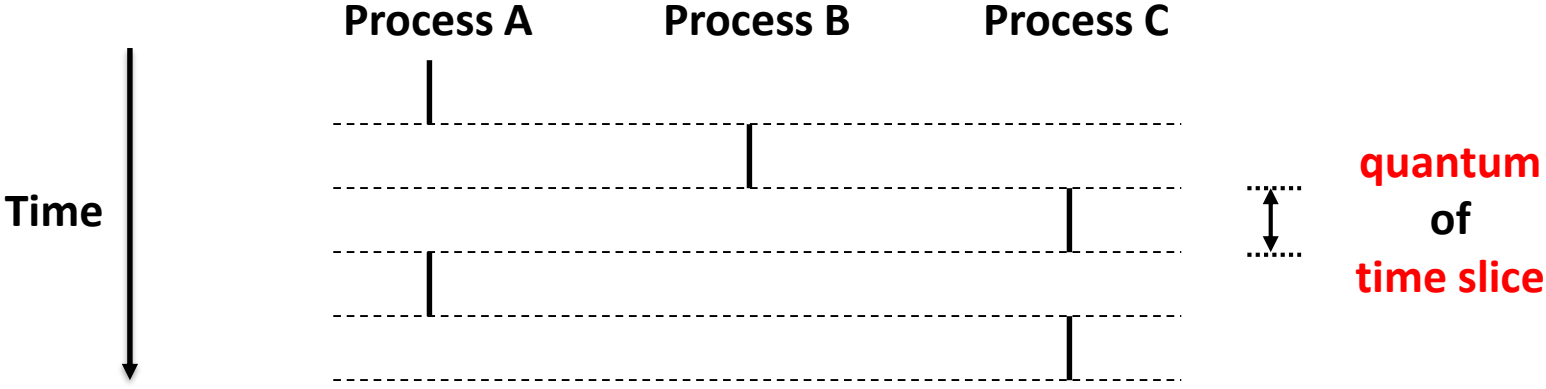
Dong-kun Shin
Embedded Software Laboratory
Sungkyunkwan University
<http://nyx.skku.ac.kr>

Processes

- **An instance of a program in execution.**
 - One of the most profound ideas in computer science.
 - Not the same as “program” or “processor”.
- **Process provides each program with two key abstractions:**
 - Logical control flow
 - Each program seems to have exclusive use of the CPU.
 - Private address space
 - Each program seems to have exclusive use of main memory.
- **How are these illusions maintained?**
 - Process executions interleaved (multitasking).
 - Address space managed by virtual memory system.

Logical Control Flows

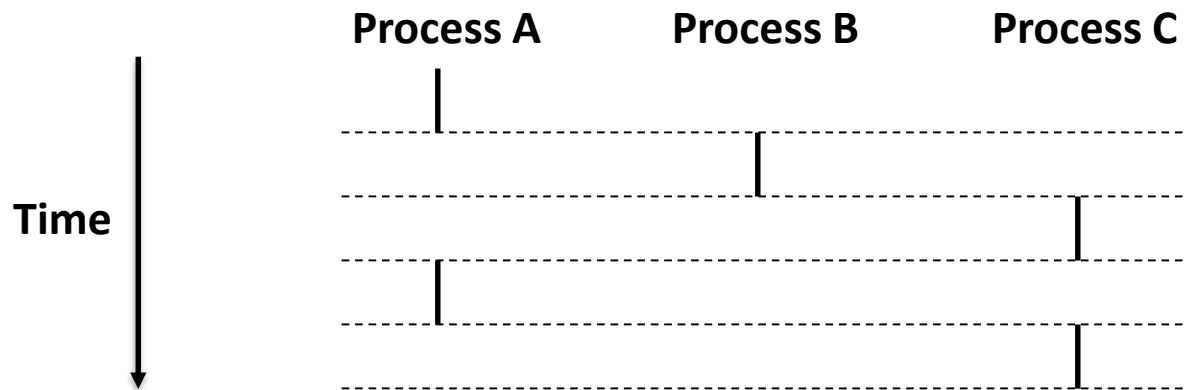
- Each process has its own logical control flow.



Concurrent Processes (1)

- **Definition**

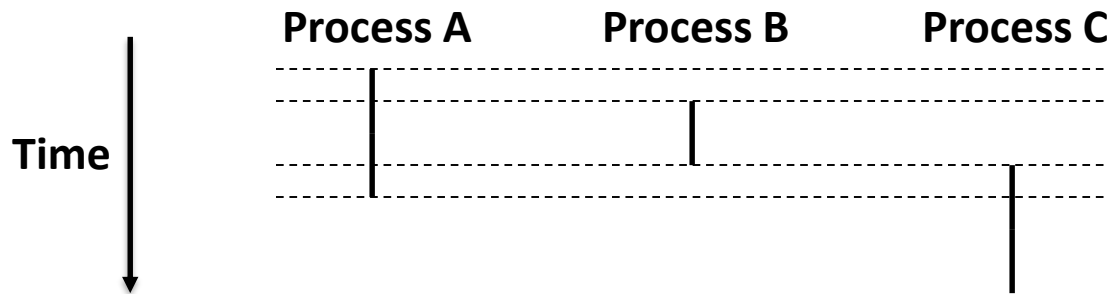
- Two processes **run concurrently** (are concurrent) if their flows overlap in time.
- Otherwise, they are **sequential**.
- Examples (running on single core):
 - Concurrent: A & B, A & C
 - Sequential: B & C



Concurrent Processes (2)

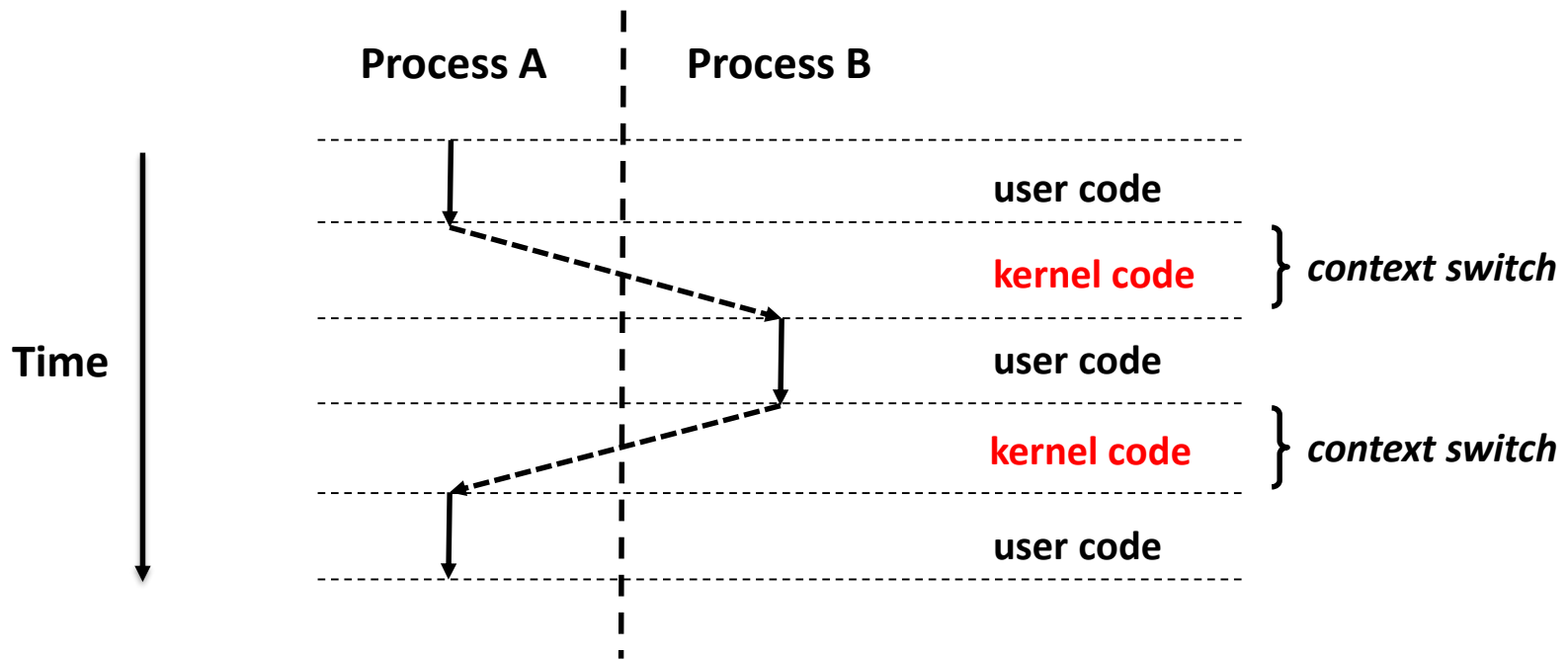
- **User View of Concurrent Processes**

- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes are running in parallel with each other



Context Switching

- Control flow passes from one process to another via a **context switch**



Creating a New Process

- **pid_t fork(void)**

- Creates a new process (child process) that is identical to the calling process (parent process)
- Returns 0 to the child process
- Returns child's **pid** to the parent process

```
if (fork() == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

**Fork is interesting
(and often confusing)
because it is called
once but returns *twice***

Fork Example (1)

- **Key points**

- Parent and child both run same code.
 - Distinguish parent from child by return value from **fork()**
- Start with same state, but each has private copy.
 - Share file descriptors, since child inherits all open files.

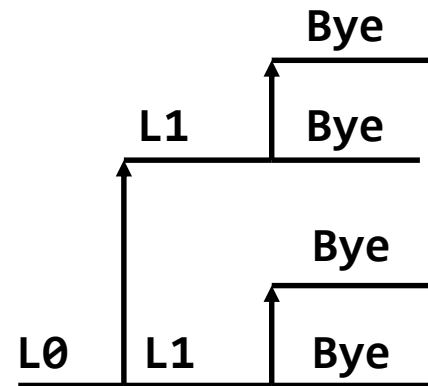
```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```


Fork Example (2)

- **Key points**

- Both parent and child can continue forking.

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Destroying a Process

- **void exit (int status)**
 - Exits a process.
 - Normally returns with status 0
 - **atexit()** registers functions to be executed upon exit.

```
void cleanup(void) {  
    printf("cleaning up\n");  
}  
  
void fork3() {  
    atexit(cleanup);  
    fork();  
    exit(0);  
}
```

Zombies (1)

- **Idea**

- When a process terminates, still consumes system resources.
 - Various tables maintained by OS
- Called a “zombie”
 - Living corpse, half alive and half dead

- **Reaping**

- Performed by parent on terminated child.
- Parent is given exit status information.
- Kernel discards the terminated process.

- **What if parent doesn't reap?**

- If any parent terminates without reaping a child, then child will be reaped by `init` process.
- Only need explicit reaping for long-running processes.
 - e.g. shells and servers

Zombies (2)

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6639 ttyp9        00:00:03 forks
 6640 ttyp9        00:00:00 forks <defunct>
 6641 ttyp9        00:00:00 ps
linux> kill 6639
[1] Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9        00:00:00 tcsh
 6642 ttyp9        00:00:00 ps
```

```
void fork4()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
            getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
            getpid());
        while (1); /* Infinite loop */
    }
}
```

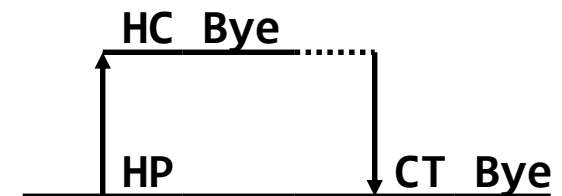
- **ps** shows child processes as "defunct"
- Killing parent allows child to be reaped

Synchronizing with Children

- **pid_t wait (int *status)**
 - suspends current process until one of its children terminates.
 - return value is the **pid** of the child process that terminated.
 - if **status != NULL**, then the object it points to will be set to a status indicating why the child process terminated.
- **pid_t waitpid (pid_t pid, int *status, int options)**
 - Can wait for specific process
 - Various options

Wait Example (1)

```
void fork5() {  
    int child_status;  
  
    if (fork() == 0) {  
        printf("HC: hello from child\n");  
    }  
    else {  
        printf("HP: hello from parent\n");  
        wait(&child_status);  
        printf("CT: child has terminated\n");  
    }  
    printf("Bye\n");  
    exit();  
}
```



Wait Example (2)

- **If multiple children completed,**
 - will take in arbitrary order.
 - Can use macros **WIFEXITED** and **WEXITSTATUS** to get information about exit status.

```
void fork6() {
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

Waitpid Example

```
void fork7()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```


Running New Programs (1)

- **int execl (char *path, char *arg0, ..., NULL)**
 - loads and runs executable at **path** with arguments **arg0**, **arg1**, ...
 - **path** is the complete path of an executable
 - **arg0** becomes the name of the process
 - Typically **arg0** is either identical to **path**, or else it contains only the executable filename from path.
 - “real” arguments to the executable start with **arg1**, etc.
 - list of args is terminated by a **(char *) 0** argument.
 - returns **-1** if error, otherwise doesn't return!
- **int execv (char *path, char *argv[])**
 - argv : null terminated pointer arrays

Running New Programs (2)

- Example: running `/bin/ls`

```
main() {  
    if (fork() == 0) {  
        execl("/bin/ls", "ls", "/", 0);  
    }  
    wait(NULL);  
    printf("completed\n");  
    exit();  
}
```

```
main() {  
    char *args[] = {"ls", "/", NULL};  
    if (fork() == 0) {  
        execv("/bin/ls", args);  
    }  
    wait(NULL);  
}
```

Summary

- **Process abstraction**
 - Logical control flow
 - Private address space

- **Process-related system calls**
 - `fork()`
 - `exit()`
 - `wait()`, `waitpid()`
 - `execl()`, `execle()`, `execv()`, `execve()`, ...

Lab. Exercise

- **Make the following program:**
 - main
 - fork()
 - if child
 - exec()
 - [`$ ls -al /sys/kernel/debug`]
 - if parent
 - wait()
 - get the child's return value
 - print the child's pid and the value