SUNG KYUN KWAN UNIVERSITY

Process

Prof. Joonwon Lee (joonwon@skku.edu)
TA – Jaehyung Park (jaeseanpark@gmail.com)
TA – Luke Albano (lukealbano@arcs.skku.edu)
Sungkyunkwan University

Process (1)

- It is not same as "program" or "processor"

- What's the difference between ...
  - Program
  - **Process**
  - Processor

- Process
  - **An instance of a program in execution**
  - One of the most profound ideas in computer science

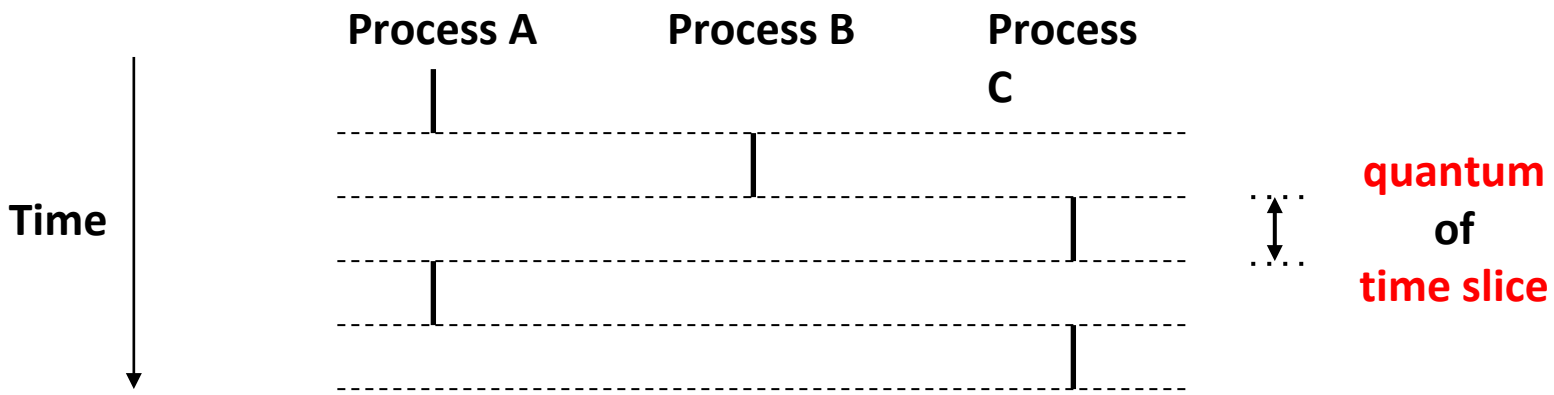
Process (2)

- 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

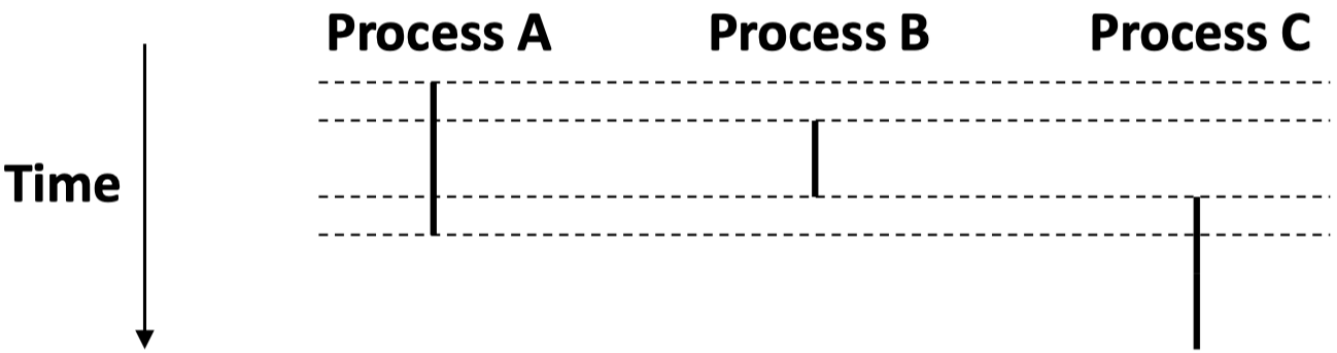
Logical Control Flows (1)

- Each process has its own logical control flow



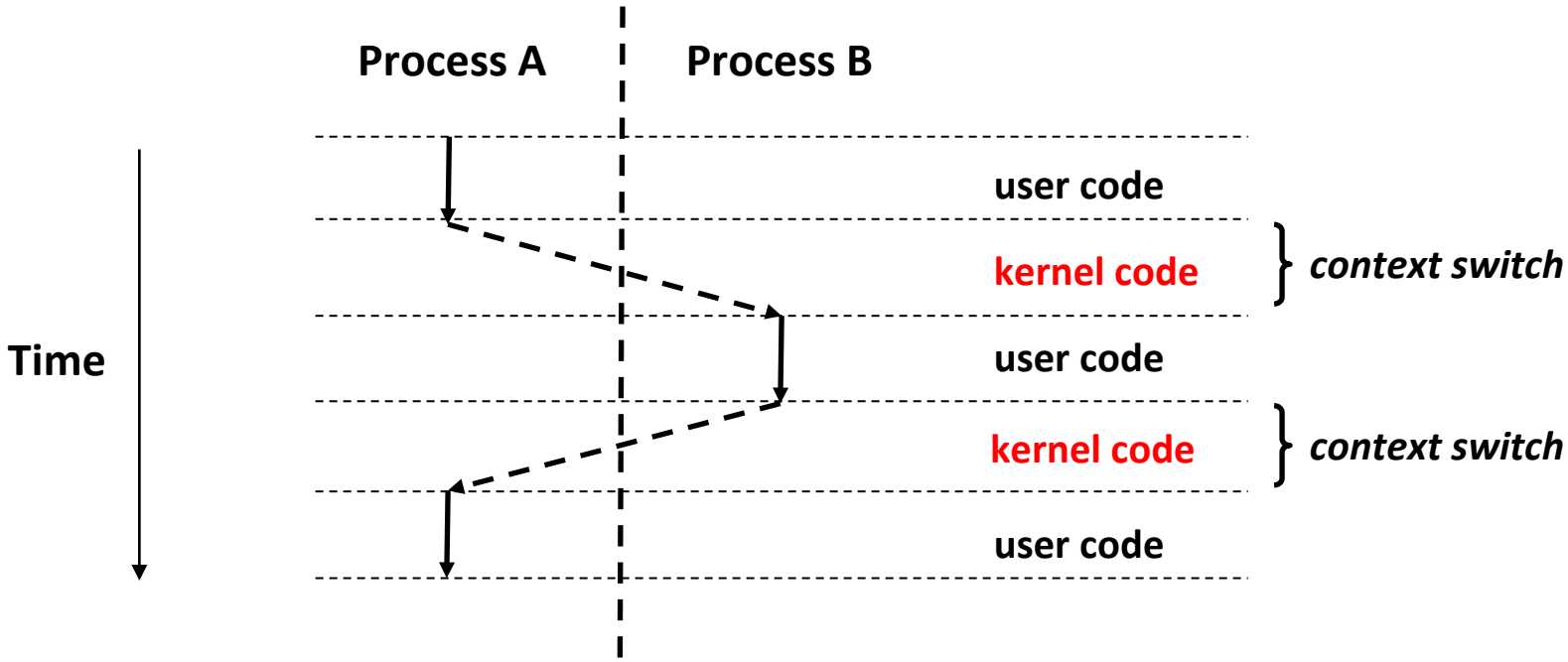
Logical Control Flows (2)

- 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



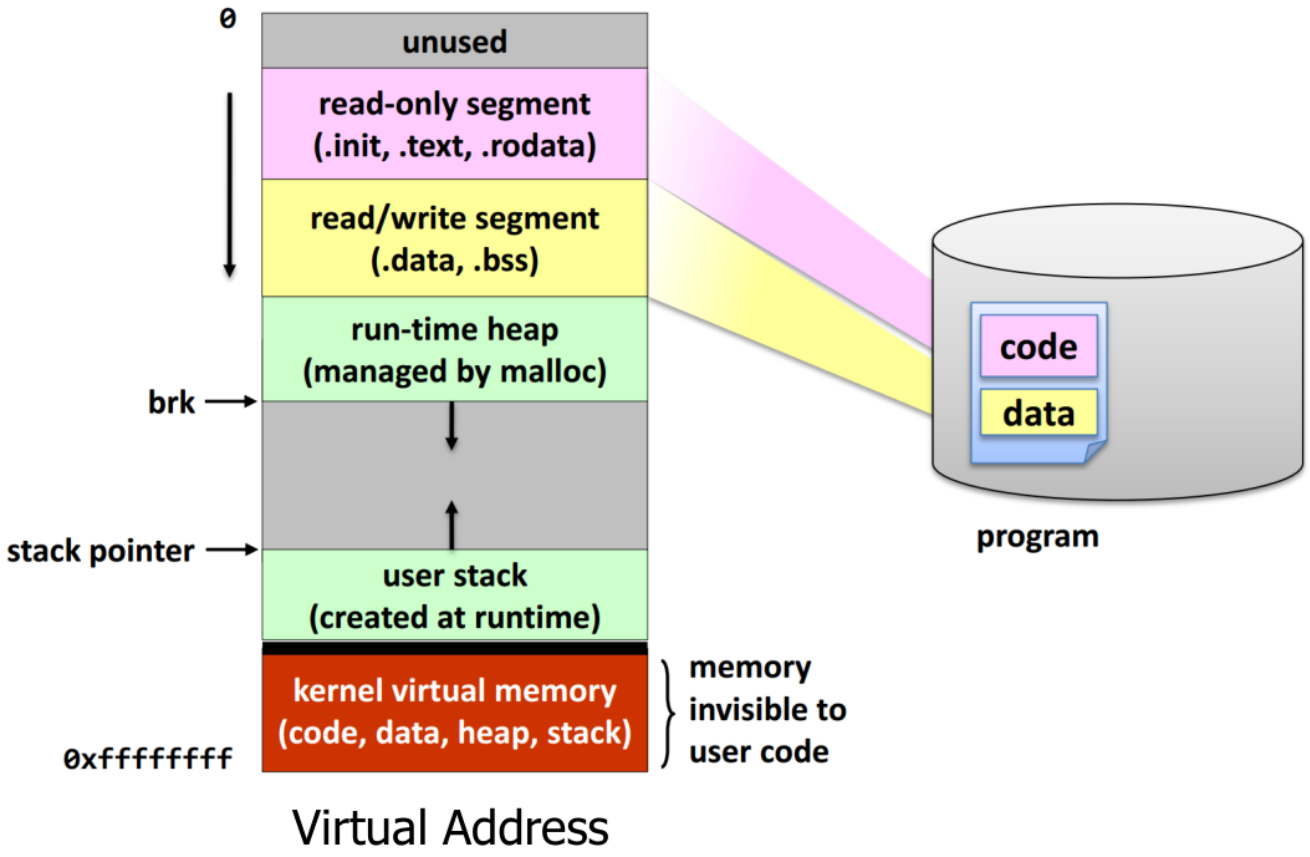
Process Control Block

- A data structure used by OS to store all the information for efficient process management

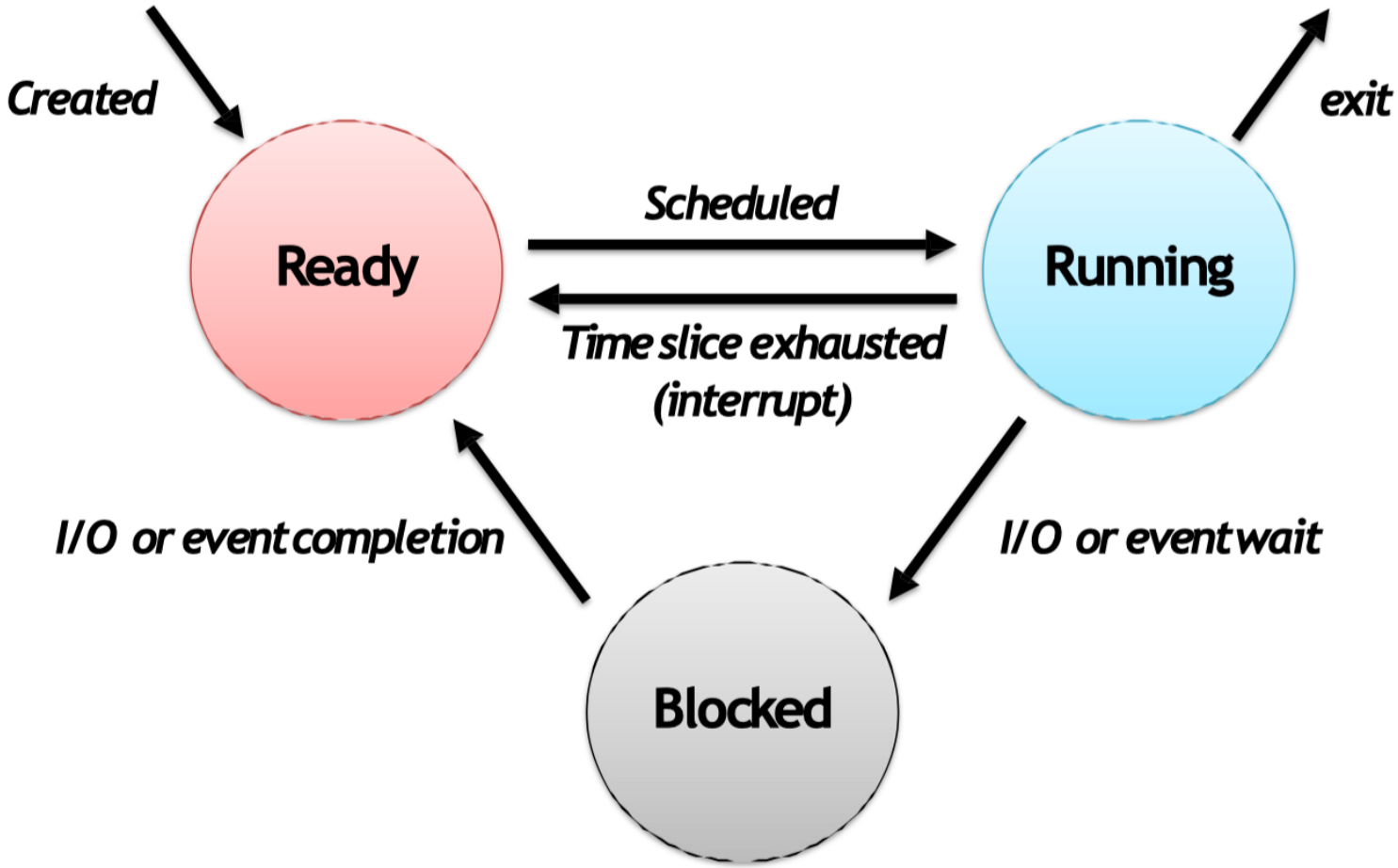
Process ID
Process State
Program Counter
Registers
Memory Info.
List of Open Files
Process Priority
I/O Status, etc…

Private Address Space

- Process in memory

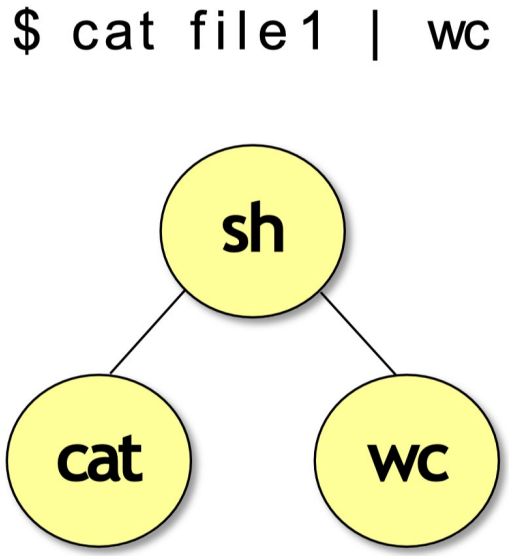


Process State Transition



Process Hierarchy

- Parent-child relationship
  - One process can create another process
  - Unix calls the hierarchy a "process group"
  - Windows has no concept of process hierarchy
- Browsing a list of processes:
  - *ps* in Unix
  - Task manager (*taskmgr*) in Windows



Creating a New Process

- **pid_t fork (void)**
  - Create a new process (child process) that is identical to the calling process (parent process)
  - Return 0 to the child process
  - Return 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)

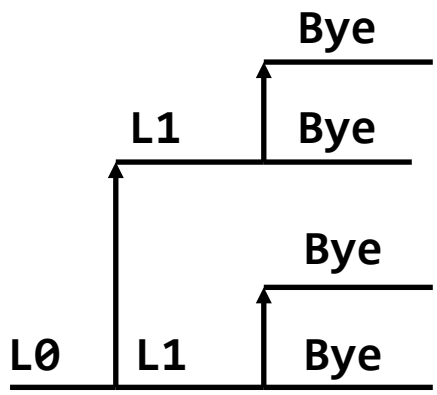
- Key points
  - Parent and child both run **same code**
    - Distinguish parent from child by **return value** from **fork()**
  - Start with same state, but each has private copy
    - Share file descriptors, since child inherits all open files

```
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)**
  - Exit 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);
}
```

Synchronizing with Children

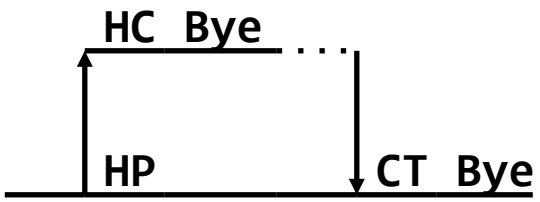
- **pid_t wait (int *status)**
  - Suspend the current process until one of its children terminates
  - Return value is the **pid** of the child process terminated
  - if **status != NULL**, then the object it points to will be set to a status indicating why the child process terminated

- **pid_t waitpid (pid_t pid, int *status, int options)**
  - Can wait for specific process
  - Various options

Wait Example (1)

```
void fork4()
{
   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 fork5()
{
    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 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 = 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);
    }
}
```

Process Termination

- Normal / error exit (voluntary)
- Fatal error (involuntary)
  - Segmentation fault (illegal memory access)
  - Protection fault
  - Exceed allocated resources, etc.
- Killed by another process (involuntary)
  - By receiving a signal
- Zombie process: terminated, but not removed

Zombie (1)

- Zombie process
  - Living corpse, half alive and half dead
  - When a process terminates, still consumes system resources
    - Various tables maintained by OS

- 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

Zombie (2)

```
void fork7()
{
  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 */
  }
}
```

```
linux> ./fork7 &
[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 fork7
 6640 ttyp9    00:00:00 fork7 <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
```

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

Running New Programs (1)

- **int execl (char *path, char *arg0, ..., NULL)**
  - Load and run 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 NULL
  - Return -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", "/", NULL);
   }
   wait(NULL);
   printf("completed\n");
   exit();
}
```

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

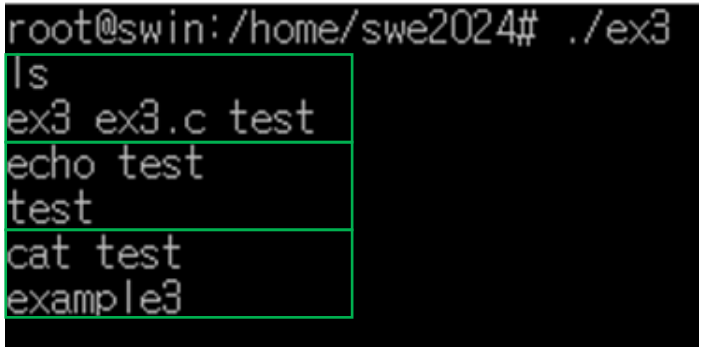
Summary

- Process abstraction
  - Logical control flow
  - Private address space

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

Exercise

- Make simple mini shell
  - This program should be under an infinite loop with conditional exit ("quit")
  - When the command is entered, the command is executed using the child process
  - When the program is quitted, the parent process must wait for all the child processes to be done before exiting itself
  - The mini shell only executes programs under /bin directory
  - There is no limit to the header file



Skeleton code of p5.c

- copy the skeleton code to your directory

$ cp ~swe2024-41_23s/2023s/p5_skeleton.c./

```
#!/bin/bash

# 1) compile your program with makefile created in Exercise 1
# your code here

# 2) make answer sheet as answer.txt
# your code here

# 3) run your program and save output as output.txt
echo "22 11" | ./p2.out > output.txt

# 4) compare answer.txt and output.txt with diff command as result.txt
# your code here

# 5) print the contents of "result.txt"
# your code here

# 6) clean object files and executable file
# your code here
```

Exercise

- Submit your Makefile & p5.c as p5
  - InUiYeJi Cluster
  - Submit the folder into p5
  - ~swe2024-41_23s/bin/submit p5 p5
  - Due

```
2014311240@swji:/home/2014311240$ ~swe2024-41_23s/bin/submit p5 p5
user name           :2014311240
Submitted Files for p5:
File Name                                   File Size       Time
--------------------------------------------------------------------------------
p5-2014311240-Mar.29.17.12.153059682        730             Wed Mar 29 17:12:53 2023
2014311240@swji:/home/2014311240$
```

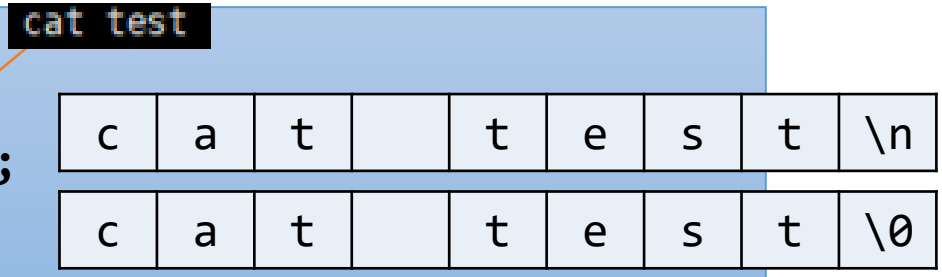
./p5

- Makefile
- p5.c

Exercise Hint

- *getline* function

```
char *cmd;
size_t size;
getline(&cmd, &size, stdin);
cmd[strlen(cmd)-1] = '\0';
free(cmd);
```



- *strtok* function

```
int i = 0;
char *ptr = strtok(cmd, " ");

while (ptr != NULL) {
        arg[i++] = ptr;
        ptr = strtok(NULL, " ");
}
arg[i] = NULL;
```

Exercise Hint

- *sprintf()* function

```
char path[100];
sprintf(path, "/bin/%s", arg[0]);
```

/	b	i	n	/	c	a	t	\0

  - Alternative: *snprintf()* for safety

Summary Report

- Summary report about man command result of
  - **`fork()`**
  - **`exit()`**
  - **`wait()`**
  - **`execv()`**

- Submission form
  - A4 size PDF format (No page limitation)
  - [SWE2024 Report-5] studentID_name
  - Ex) [SWE2024 Report-5] 2022XXXXXX_박재형
  - Submit to iCampus
  - Due: until Friday, 31 March 2023, 23:59