Chapter 3

Processes
(Part 1)

Yunmin Go

School of CSEE

HANDONG GLOBAL UNIVERSITY

Agenda

- Process Concept
- Process State and Scheduling
- Process API
- Signal
- Inter Process Communication

Virtualizing CPU

- The OS can promote the illusion that many virtual CPUs exist.

cpu.c

```c
int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];

    while (1) {
        printf("%s\n", str);
        Spin(1);
    }
    return 0;
}
```

```
yunmin@yunmin:~/workspace/ch3$ ./cpu A & ./cpu B & ./cpu C &
[1] 5038
[2] 5039
[3] 5040
A
yunmin@yunmin:~/workspace/ch3$ B
C
B
A
B
C
B
C
A
B
A
C
C
A
B
B
C
A
```

Process

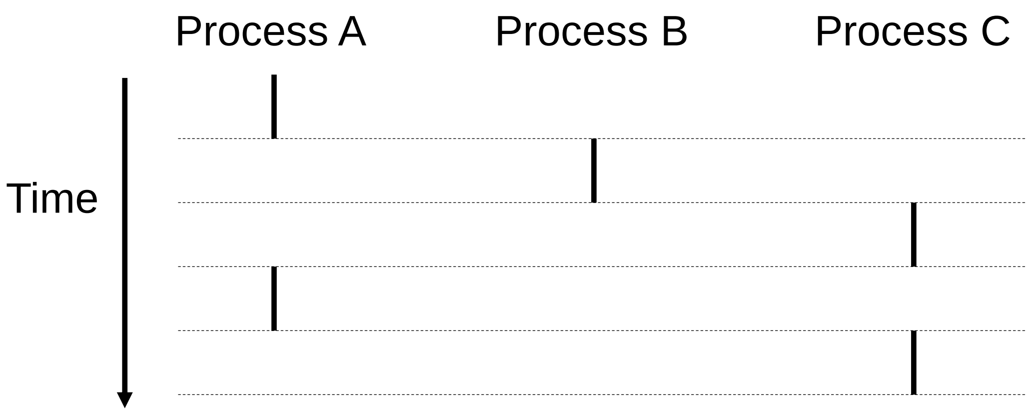
- **Process**: instance of a running program
  - Not the same as “program” or “processor”
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.

- How are these illusions maintained?
  - Interleaved execution of processes for multitasking
  - Address spaces managed by virtual memory system

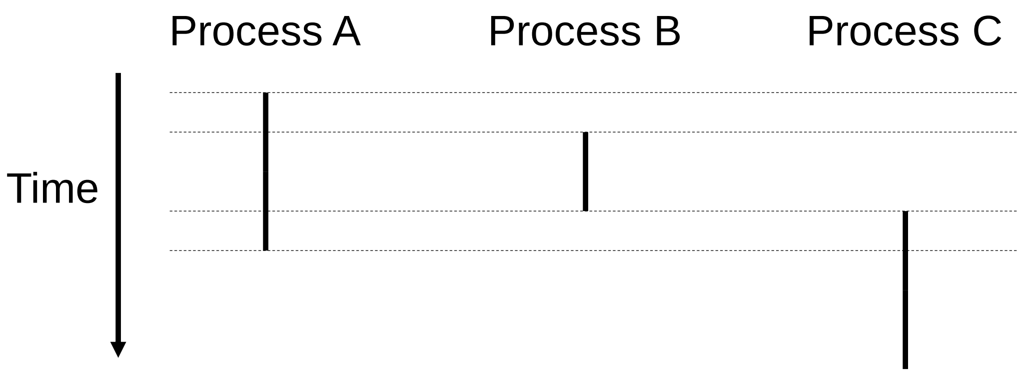
Concurrent Processes

- Two processes *run concurrently* (*are concurrent*) if their flows overlap in time.
  - Otherwise, they are sequential.
- Control flows for concurrent processes are physically disjoint in time.
- However, we can think of concurrent processes are running in parallel with each other.

<System view of concurrent processes>

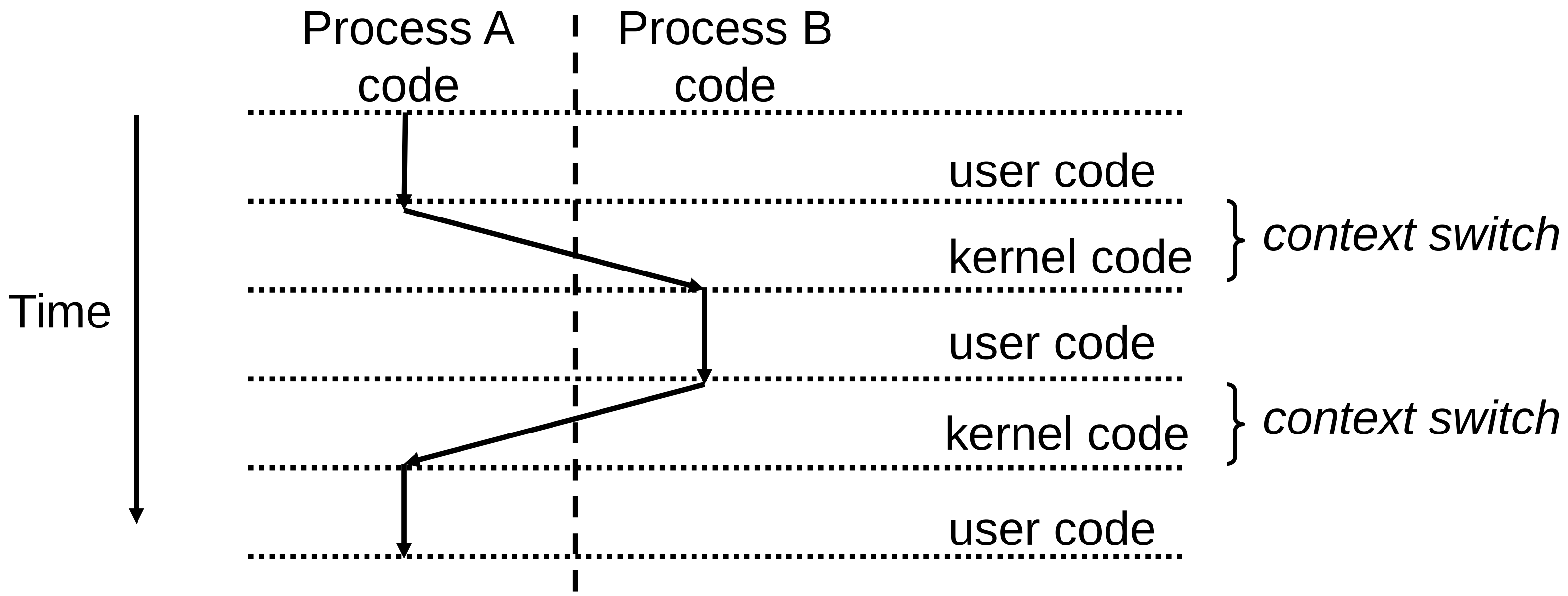


<User view of concurrent processes>



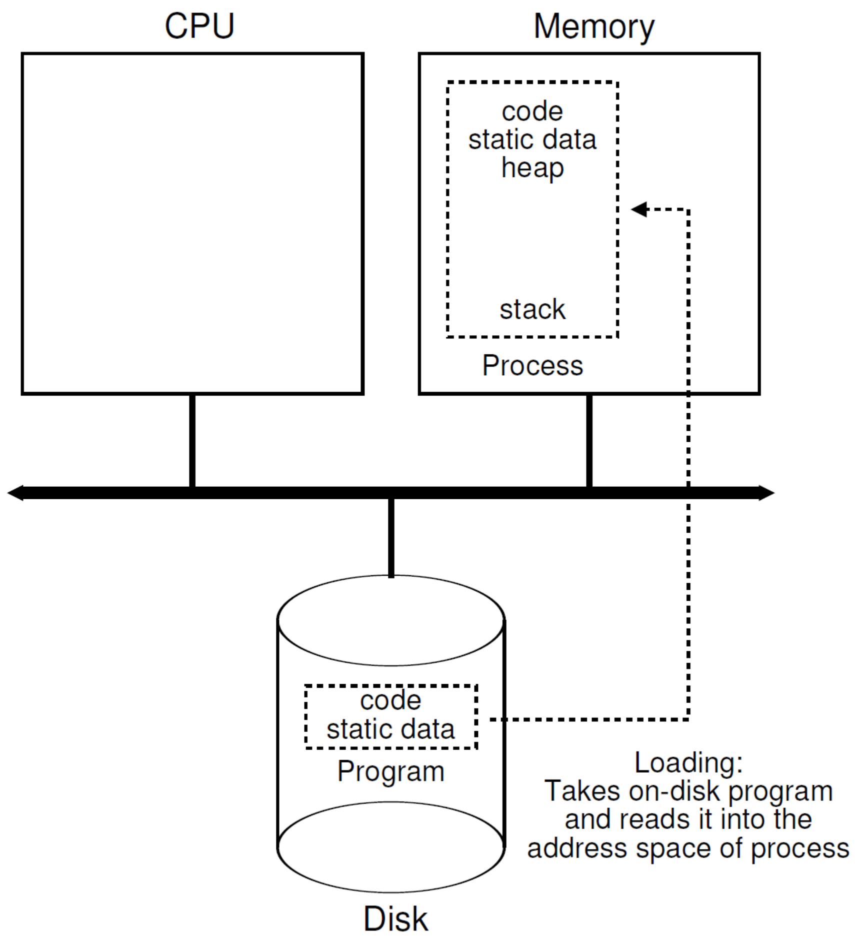
Context Switching

- Processes are managed by kernel
- Control flow passes from one process to another via a **context switch**.



Loading: From Program to Process

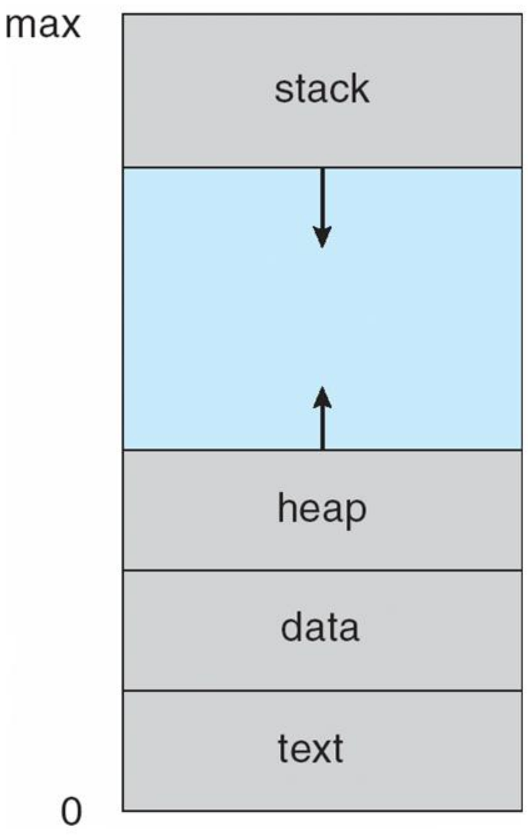
- 1. Load a program code into memory, into the address space of the process (**text**).
  - Programs initially reside on disk in executable format.
  - OS perform the loading process lazily.
    - Loading pieces of code or data only as they are needed during program execution.
- 2. The program's run-time **stack** is allocated.
  - Use the stack for local variables, function parameters, and return address.
  - Initialize the stack with arguments
    → `argc` and the `argv` array of `main()` function



HANDONG GLOBAL UNIVERSITY

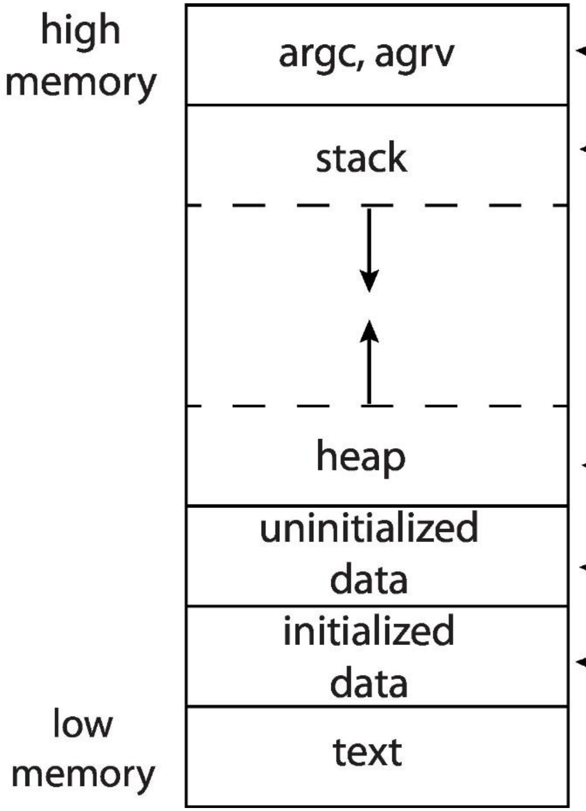
Loading: From Program to Process

- 3. The program's **heap** is created.
  - Used for explicitly requested dynamically allocated data.
  - Program request such space by calling `malloc()` and free it by calling `free()`.
- 4. The OS does some other initialization tasks.
  - Input/output (I/O) setup
    - Each process by default has three open file descriptors → standard input, output and error
- 5. Start the program running at the entry point, namely **main()**.
  - The OS transfers control of the CPU to the newly-created process.



<Layout of a process in memory>

Memory Layout of a C program




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

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Process API

- **Create**: Create a new process to run a program
- **Destroy**: Halt a runaway process
- **Wait**: Wait for a process to stop running
- **Miscellaneous Control**: Some kind of method to suspend a process and then resume it
- **Status**: Get some status info about a process

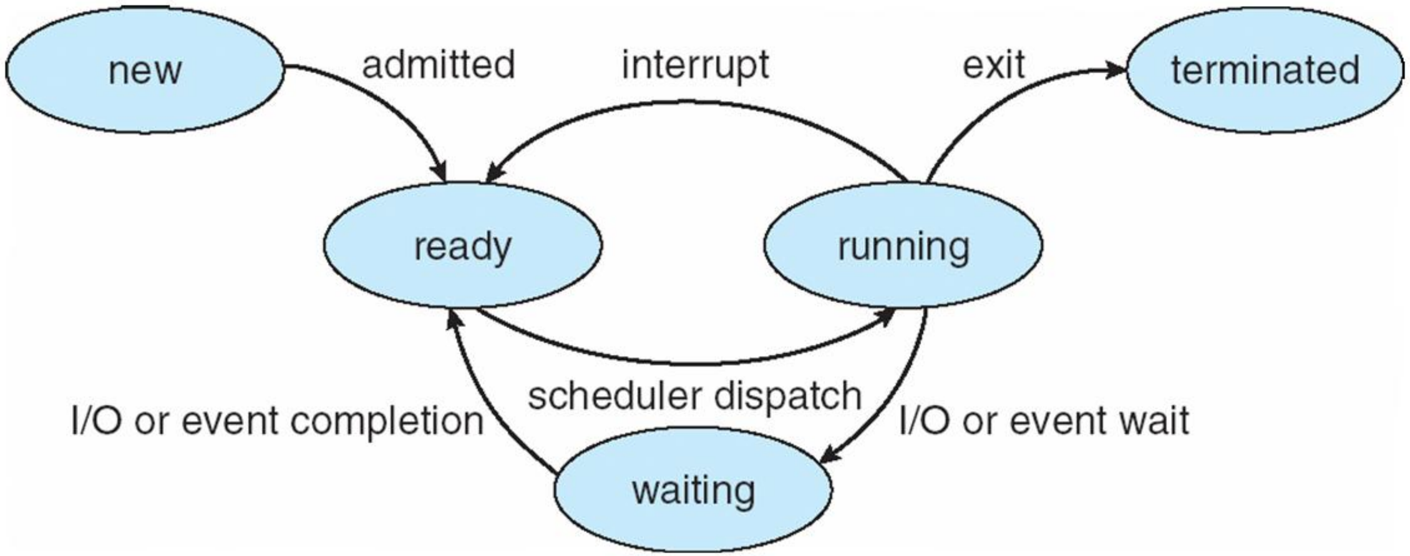
- Above APIs are available on any modern OS.

Agenda

- Process Concept
- **Process State and Scheduling**
- Process API
- Signal
- Inter Process Communication

Process State

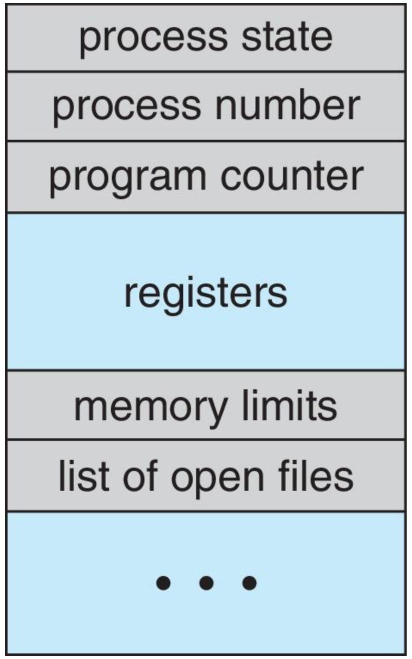
- As a process executes, it changes state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution



Process Control Block (PCB)

- OS manages processes using PCB
  - Process Control Block (PCB): Information associated with each process
    - also called task control block

Category	Information
Process state	new, ready, running, waiting, terminated, …
Process number	pid (Process ID)
CPU Registers	program counter, accumulator, general registers, stack pointer, …
CPU Scheduling info.	priority, scheduling queue pointers
Memory-management info.	base and limit registers, page/segment table, …
Accounting info.	CPU-time used, clock time elapsed since start, time limits, …
I/O status info.	I/O devices allocated to process, List of open files



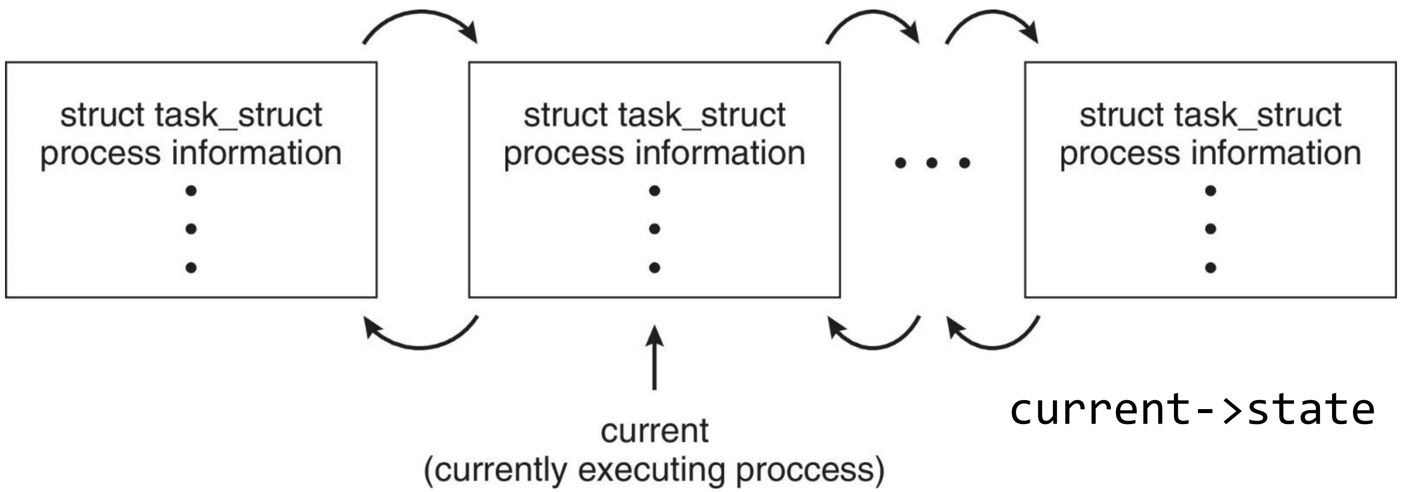
<PCB>

Process Representation in Linux

- Represented by the C structure `task_struct`

  `<include/linux/sched.h>` in the kernel source-code

```
pid t_pid;                     /* process identifier */
long state;                    /* state of the process */
unsigned int time_slice        /* scheduling information */
struct task_struct *parent;    /* this process's parent */
struct list_head children;     /* this process's children */
struct files_struct *files;    /* list of open files */
struct mm_struct *mm;          /* address space of this process */
```



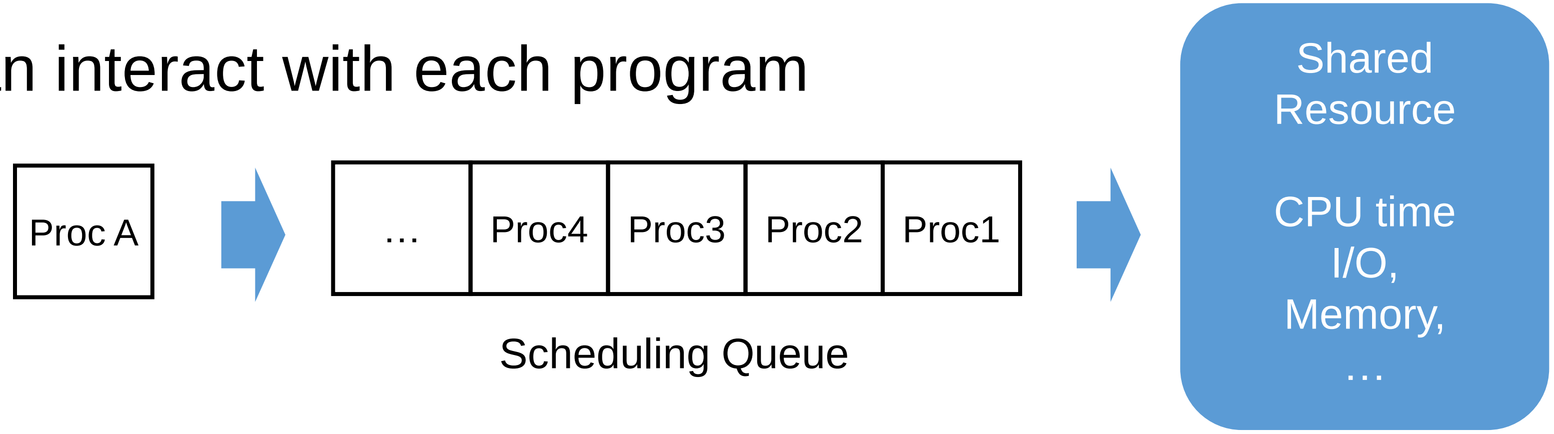
`current->state = new_state`

Threads

- Thread: a way for a program to split itself into two or more simultaneously running task
  - Most modern operating systems have extended the process concept to allow a process to have multiple thread of execution
    → can perform more than one task at a time
  - Especially beneficial on multicore systems, where multiple thread can run in parallel
  - The PCB is expanded to include information for each thread
    → multiple program counters in PCB
  - See next chapter

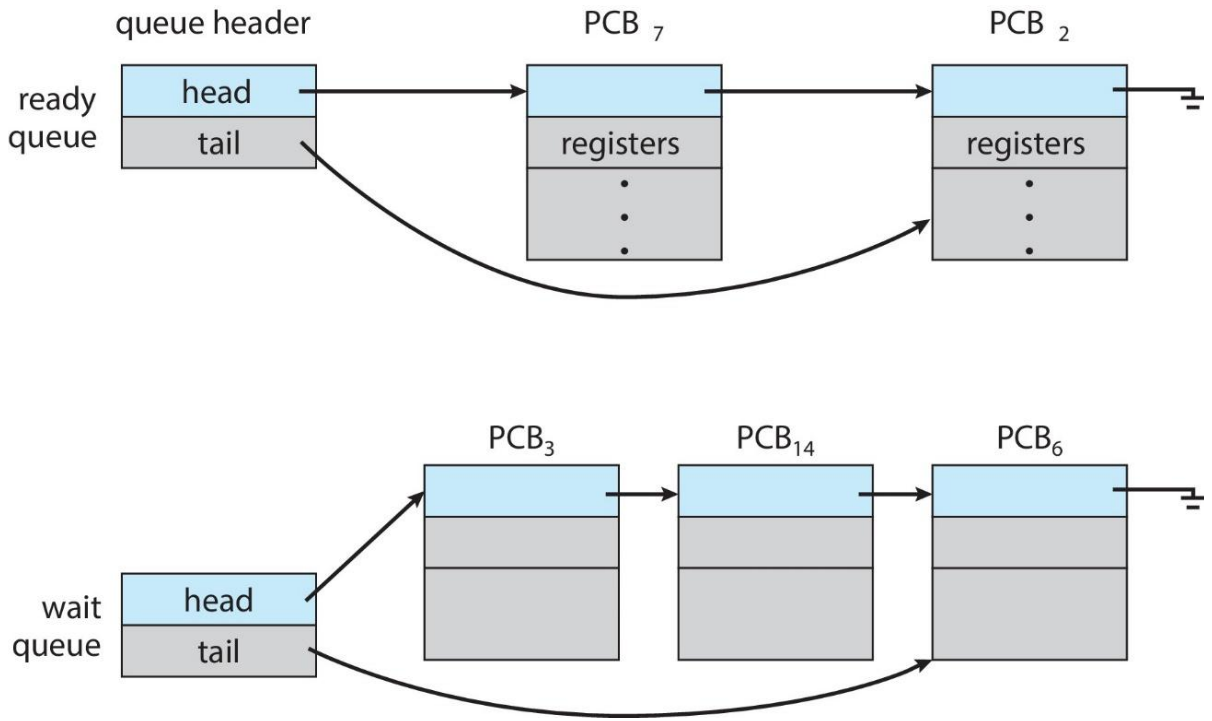
Process Scheduling

- **Process scheduling**: selecting a process to execute on CPU
  - Only one process can run on each processor at a time
  - Process scheduler selects among available processes for next execution
  - Other processes should wait in the scheduling queue
    - Scheduling queue: waiting list of processes for CPU time or other resources.
    - Degree of multiprogramming: the number of processes currently in memory
- Objectives of scheduling
  - Maximize CPU utilization
  - Users can interact with each program



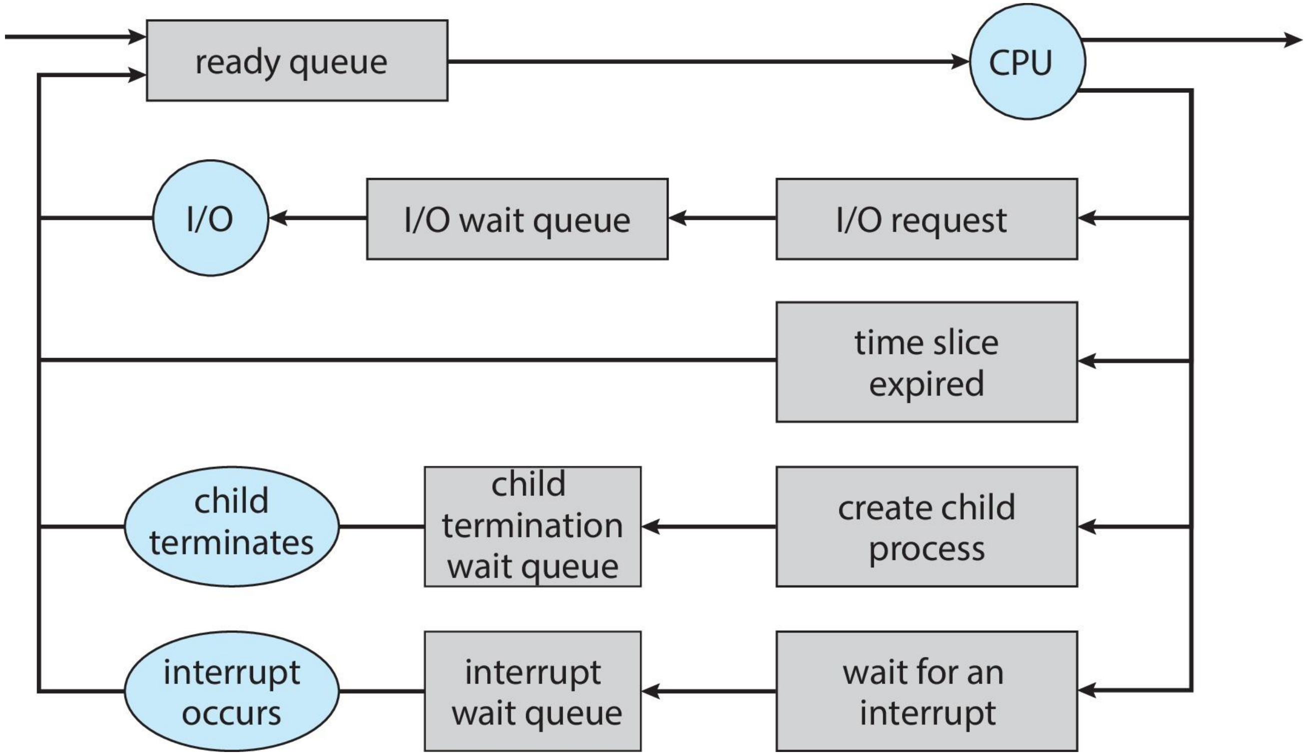
Scheduling Queues

- Types of scheduling queues
  - Ready queue: set of all processes residing in main memory, ready and waiting to execute on a CPU's core
  - Wait queue: set of processes waiting for a certain event to occur (i.e. I/O)



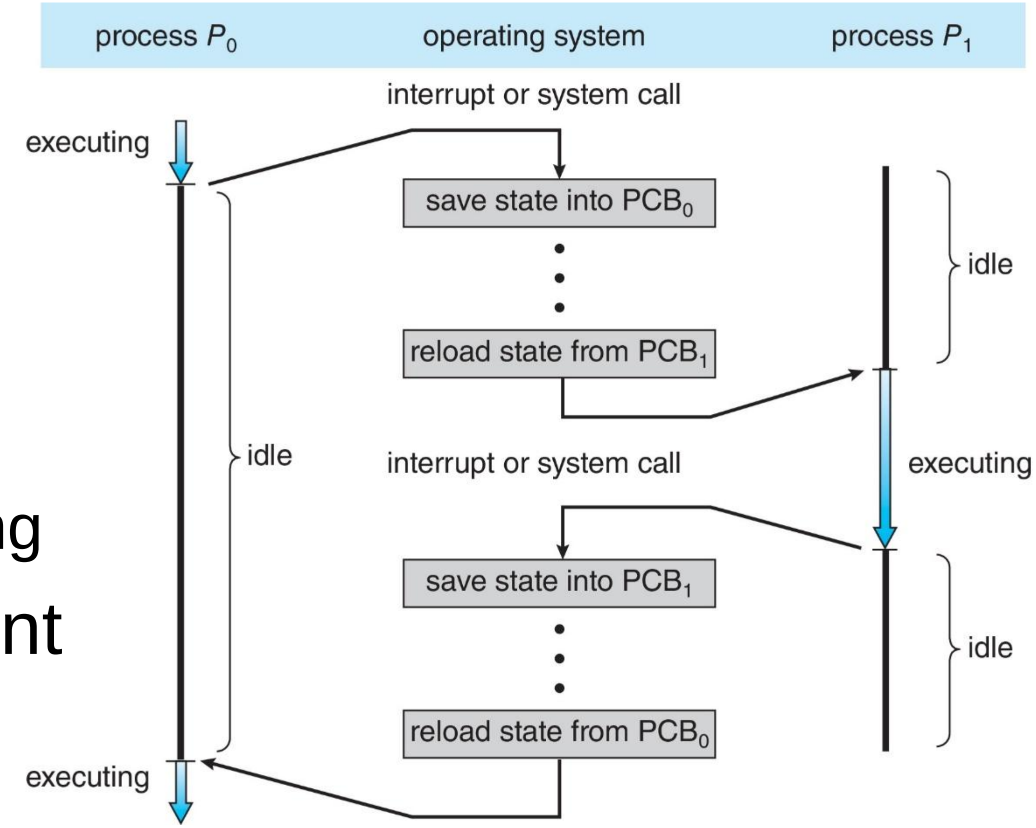
Scheduling Queues

- Representation of process scheduling
  - Processes migrate among the various queues



Context Switch

- **Context switch**: CPU switches from one process to another
  - When CPU switches to another process, save the state of the current process and load the saved state for the new process
- Context of a process represented in the PCB
- Context-switch time is overhead
  - System does not useful work while switching
- Context switch time are highly dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

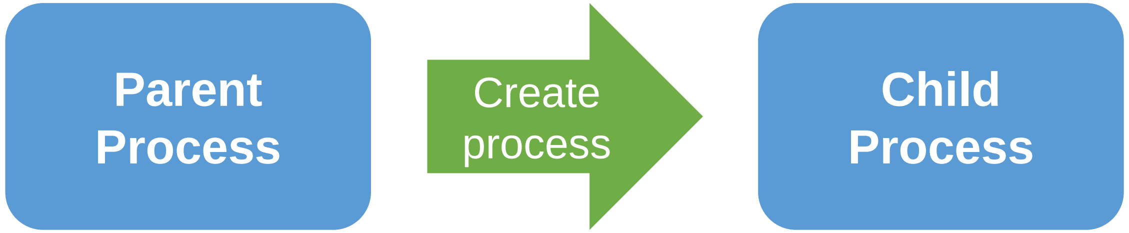


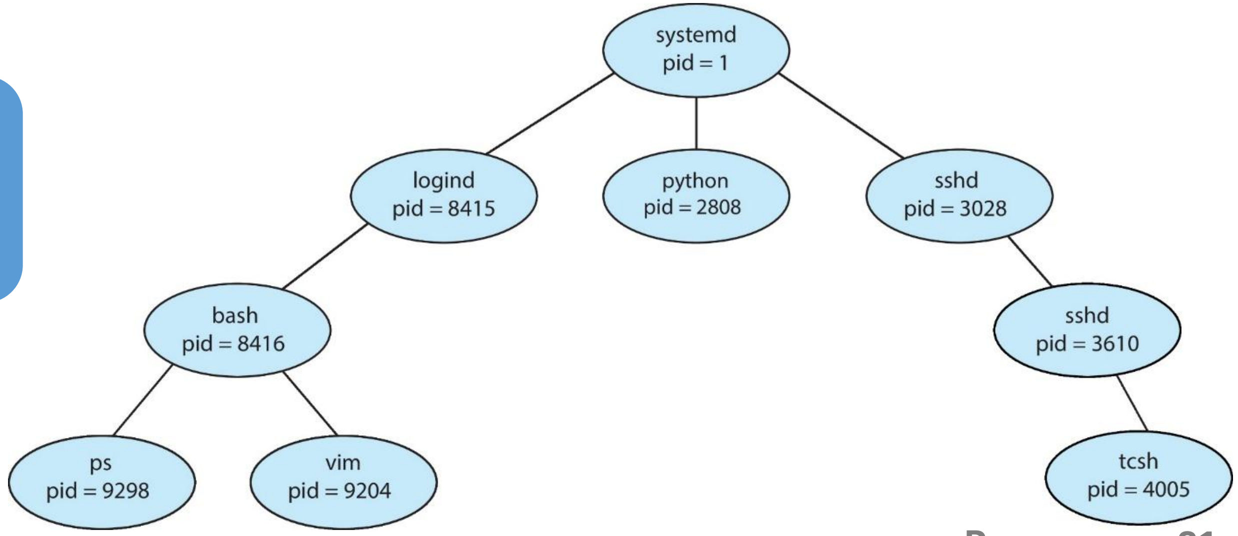
Agenda

- Process Concept
- Process State and Scheduling
- Process API
- Signal
- Inter Process Communication

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
  - Process tree: parent-child relation between processes
- Process is identified and managed via a **process identifier** (**pid**)
  - pid provides a unique value for each process in the system
  - It can be used as an index to access various attributes of a process





Displaying Process Information

- Linux
  - $ ps [-ef]

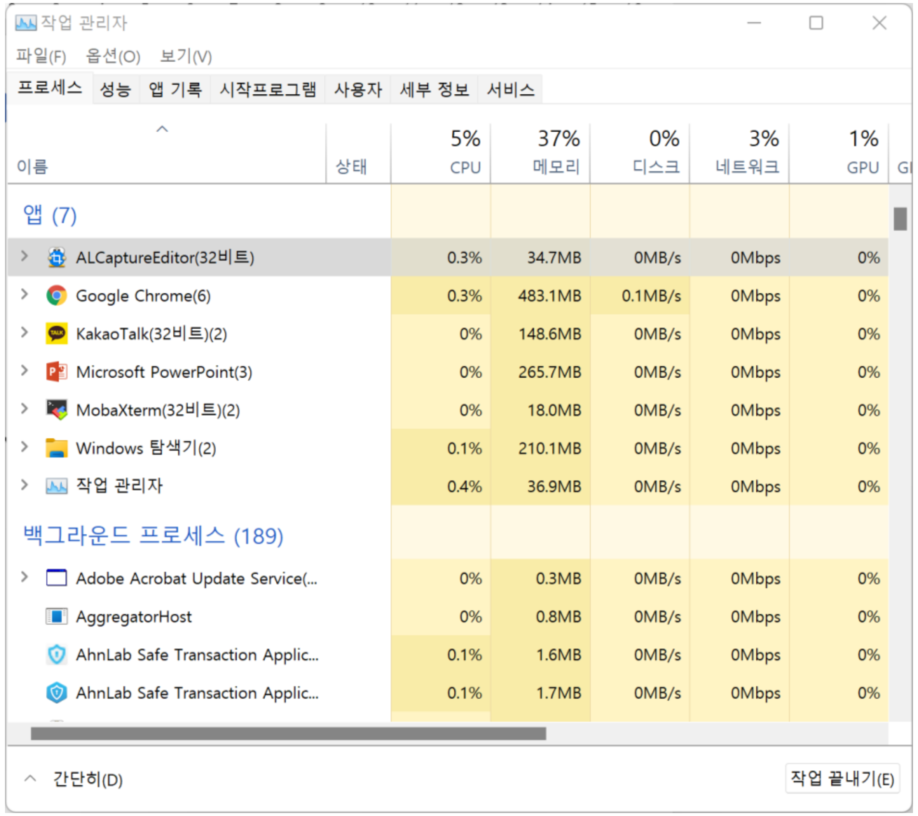
```
yunmin@yunmin:~/workspace/ch3$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 Mar19 ?        00:00:07 /sbin/init splash
root           2       0  0 Mar19 ?        00:00:00 [kthreadd]
root           3       2  0 Mar19 ?        00:00:00 [pool_workqueue_release]
root           4       2  0 Mar19 ?        00:00:00 [kworker/R-rcu_gp]
root           5       2  0 Mar19 ?        00:00:00 [kworker/R-sync_wq]
root           6       2  0 Mar19 ?        00:00:00 [kworker/R-slub_flushwq]
root           7       2  0 Mar19 ?        00:00:00 [kworker/R-netns]
root          11       2  0 Mar19 ?        00:00:00 [kworker/u8:0-ipv6_addrconf]
root          12       2  0 Mar19 ?        00:00:00 [kworker/R-mm_percpu_wq]
root          13       2  0 Mar19 ?        00:00:00 [rcu_tasks_kthread]
root          14       2  0 Mar19 ?        00:00:00 [rcu_tasks_rude_kthread]
root          15       2  0 Mar19 ?        00:00:00 [rcu_tasks_trace_kthread]
root          16       2  0 Mar19 ?        00:00:05 [ksoftirqd/0]
root          17       2  0 Mar19 ?        00:00:03 [rcu_preempt]
root          18       2  0 Mar19 ?        00:00:00 [rcu_exp_par_gp_kthread_worker/0]
root          19       2  0 Mar19 ?        00:00:00 [rcu_exp_gp_kthread_worker]
root          20       2  0 Mar19 ?        00:00:00 [migration/0]
root          21       2  0 Mar19 ?        00:00:00 [idle_inject/0]
root          22       2  0 Mar19 ?        00:00:00 [cpuhp/0]
root          23       2  0 Mar19 ?        00:00:00 [cpuhp/1]
root          24       2  0 Mar19 ?        00:00:00 [idle_inject/1]
root          25       2  0 Mar19 ?        00:00:00 [migration/1]
root          26       2  0 Mar19 ?        00:00:01 [ksoftirqd/1]
root          31       2  0 Mar19 ?        00:00:00 [kdevtmpfs]
root          32       2  0 Mar19 ?        00:00:00 [kworker/R-inet_frag_wq]
root          34       2  0 Mar19 ?        00:00:00 [kauditd]
```

  - $ pstree

```
yunmin@yunmin:~/workspace/ch3$ pstree
systemd─┬─ModemManager───3*[{ModemManager}]
        ├─NetworkManager───3*[{NetworkManager}]
        ├─accounts-daemon───3*[{accounts-daemon}]
        ├─at-spi-bus-laun─┬─dbus-daemon
        │                 └─4*[{at-spi-bus-laun}]
        ├─at-spi2-registr───3*[{at-spi2-registr}]
        ├─avahi-daemon───avahi-daemon
        ├─colord───3*[{colord}]
        ├─2*[cpptools-srv───7*[{cpptools-srv}]]
        ├─cron
        ├─cups-browsed───3*[{cups-browsed}]
        ├─cupsd
        ├─dbus-daemon
        ├─fwupd───5*[{fwupd}]
        ├─gdm3─┬─gdm-session-wor─┬─gdm-wayland-ses─┬─dbus-run-sessio─┬─dbus-dae
        │      │                 │                 │                 └─gnome-se
```

Displaying Process Information

- Windows
  - Task manager (windows system program)
  - Process explorer (freeware)



Process API in Linux

- Linux system calls related to process creation
  - **`fork()`**: create process and returns its pid
    - In parent process, return value is **pid of child**
    - In child process, return value is zero
  - **`exec()` family**: execute a program
    - The new program substitutes the original one
    - `execl()`, `execv()`, **`execlp()`**, **`execvp()`**, `execle()`, `execve()`
  - **`wait()`**: waits until child process terminates

Agenda

- Process Concept
- Process State and Scheduling
- Process API
  - Creating Process
  - Running New Program
  - Terminating and Waiting Process
- Signal
- Inter Process Communication

Creating Process: fork()

- `fork()`: Create a child process

```
#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
  - At the time of fork() both memory spaces have the same content
  - Memory writes, file mappings, and unmappings performed by one of the processes do not affect the other
  - Return value
    - Success on child process: 0
    - Success on parent process: child PID
    - Error: -1

Creating Process: Example #1

fork1.c

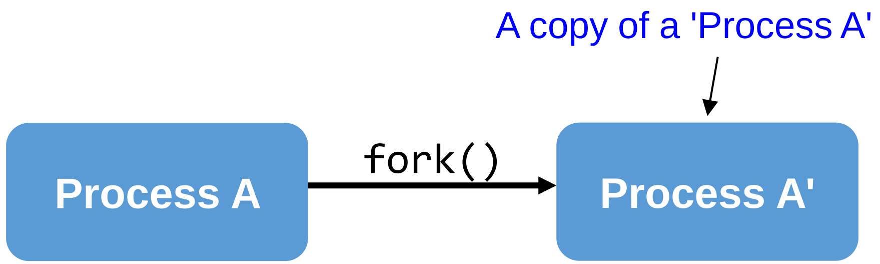
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    pid_t pid, mypid;
    printf("A\n");

    pid = fork();        /* create a new child process */
    if (pid == -1) {     /* check and handle error return value */
        printf("fork failed!\n");
        exit(pid);
    }

    if (pid == 0) {      /* the child process */
        mypid = getpid();
        printf("Child: fork returned %d, my pid %d\n", pid, mypid);
    } else  {            /* the parent process */
        mypid = getpid();
        printf("Parent: fork returned %d, my pid %d\n", pid, mypid);
    }

    printf("B:%d\n", mypid);
    return 0;
}
```



Creating Process: Example #1

- Parent process

```c
pid = fork();   // pid > 0
if (pid == -1) {
    printf("fork failed!\n");
    exit(pid);
}

if (pid == 0) {
    mypid = getpid();
    printf("Child: fork returned %d,
            my pid %d\n", pid, mypid);
} else  {
    mypid = getpid();
    printf("Parent: fork returned %d,
            my pid %d\n", pid, mypid);
}

printf("B:%d\n", mypid);
```

- Child process

```c
pid = fork();  // pid == 0
if (pid == -1) {
    printf("fork failed!\n");
    exit(pid);
}

if (pid == 0) {
    mypid = getpid();
    printf("Child: fork returned %d,
            my pid %d\n", pid, mypid);
} else  {
    mypid = getpid();
    printf("Parent: fork returned %d,
            my pid %d\n", pid, mypid);
}

printf("B:%d\n", mypid);
```

Creating Process: Example #1

fork1.c

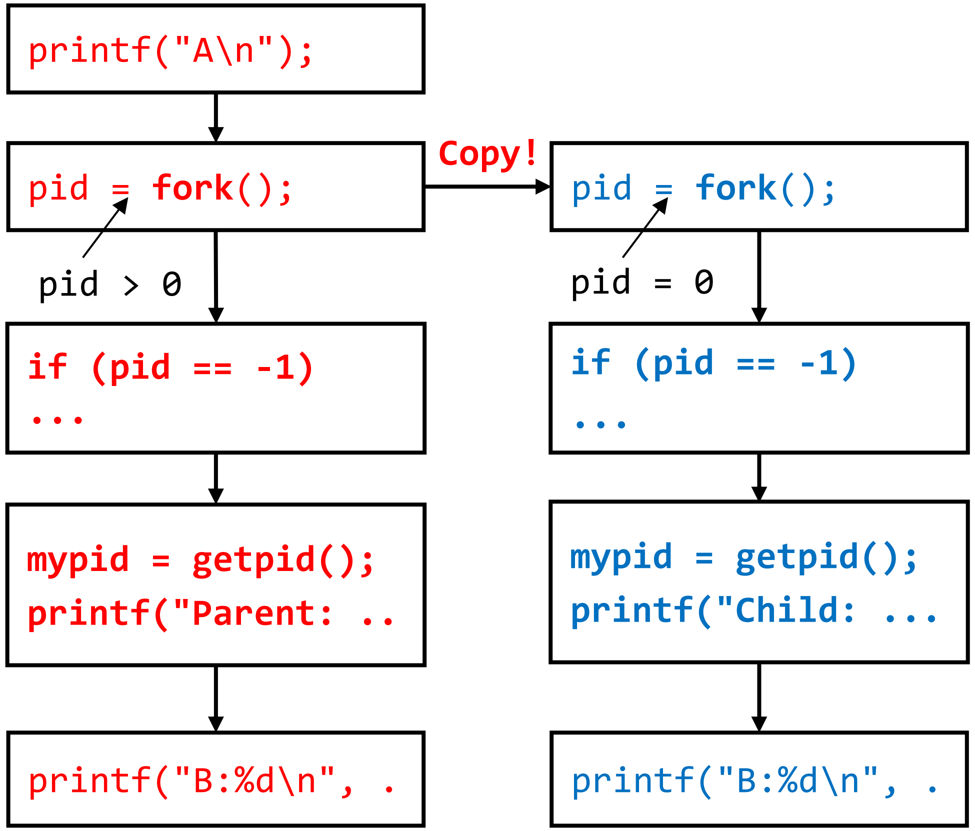
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
    pid_t pid, mypid;
    printf("A\n");

    pid = fork();          /* create a new child process */
    if (pid == -1) {       /* check and handle error return value */
        printf("fork failed!\n");
        exit(pid);
    }

    if (pid == 0) {        /* the child process */
        mypid = getpid();
        printf("Child: fork returned %d, my pid %d\n", pid, mypid);
    } else  {              /* the parent process */
        mypid = getpid();
        printf("Parent: fork returned %d, my pid %d\n", pid, mypid);
    }

    printf("B:%d\n", mypid);
    return 0;
}
```




```
yunmin@yunmin:~/workspace/ch3$ ./fork1
A
Parent: fork returned 6232, my pid 6231
B:6231
Child: fork returned 0, my pid 6232
B:6232
```

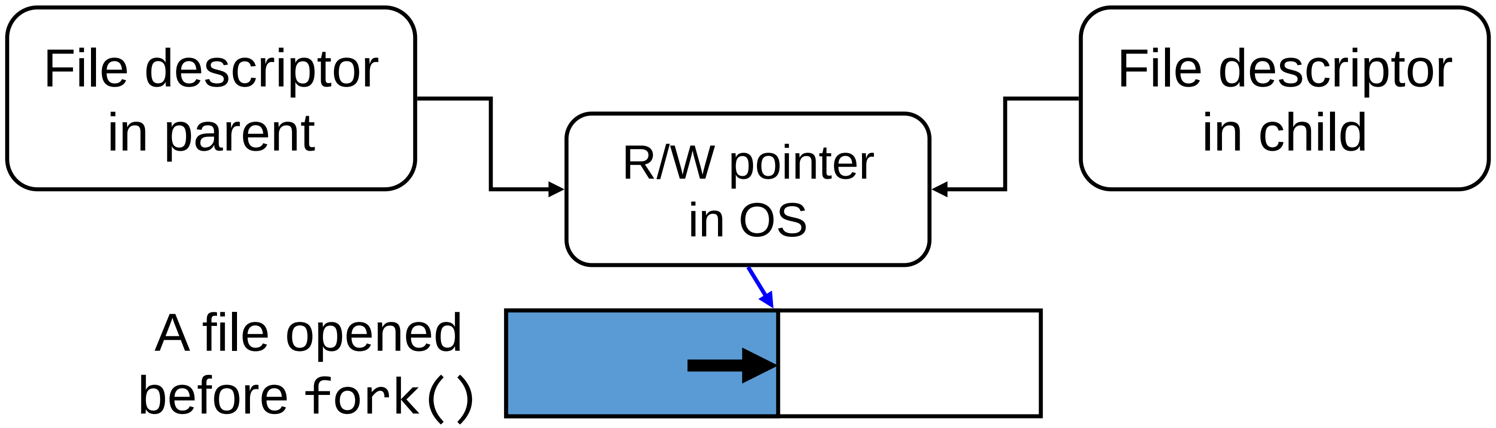
Creating Process: Example #2

- How many times will L0, L1, L2, and Bye each be printed?

```
void main()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Creating Process: Resource of Child

- Data (variables): copies of variables of parent process
  - Child process has its own address space.
  - The only difference is **the pid of child** returned from **fork().**
- Files
  - Opened before `fork()`: shared with parent
  - Opened after `fork()`: not shared



Creating Process: Example #3

fork2.c

```
int gval = 10;

int main(int argc, char *argv[])
{
  int lval = 20;
  pid_t pid;

  lval += 5;
  gval++;

  pid = fork();
  if (pid == 0)   // if Child Process
    gval++;
  else            // if Parent Process
    lval++;

  if (pid == 0)
    printf("Child Proc: [%d, %d] \n", gval, lval);
  else
    printf("Parent Proc: [%d, %d] \n", gval, lval);
  return 0;
}
```

Agenda

- Process Concept
- Process State and Scheduling
- Process API
  - Creating Process
  - Running New Program
  - Terminating and Waiting Process
- Signal
- Inter Process Communication

Running New Program: execve()

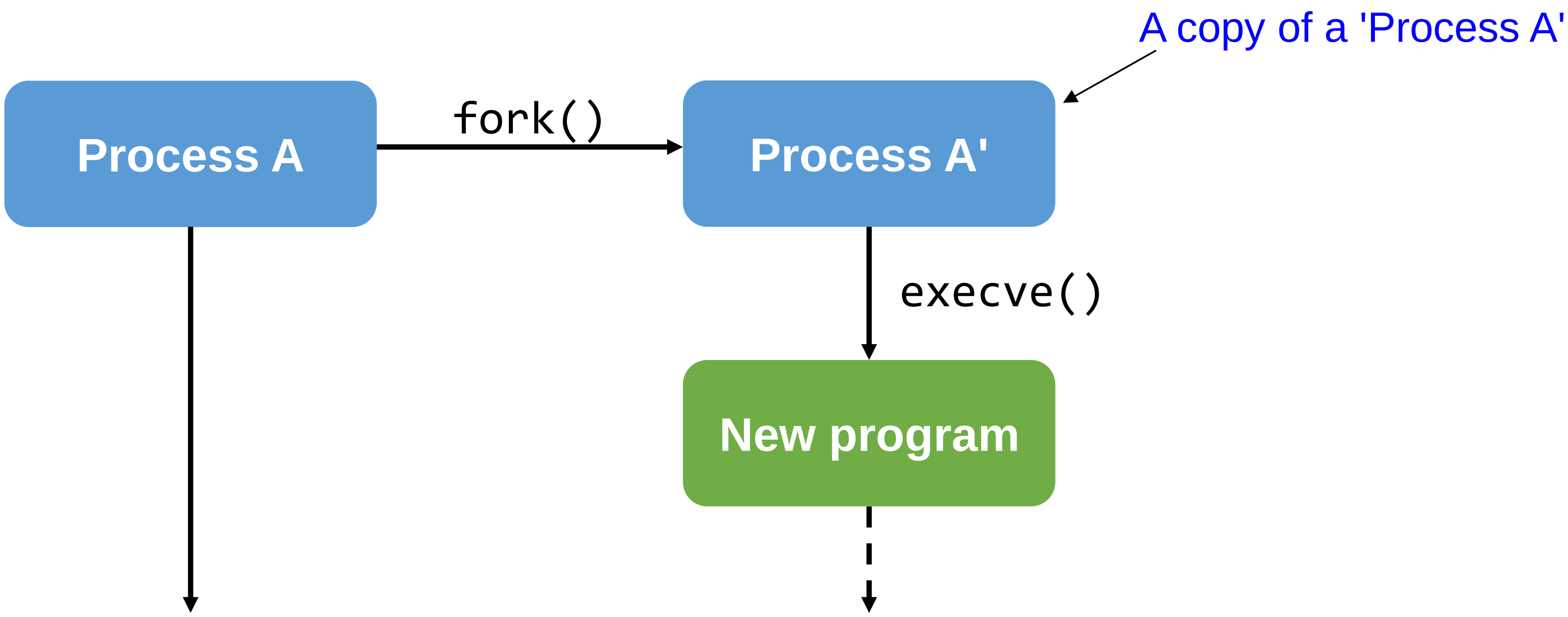
- `execve()`: execute a file

```
#include <unistd.h>
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

- Executes the program referred to by *pathname*
- Relaces the program currently running in the calling process with a new program
- *pathname*: must be either a binary executable, or a script starting with a line of the form: `#!interpreter [optional-arg]`
- *argv*: an array pointers to strings passed to the new program as its command-line arguments
- *envp*: an array of pointers to strings which are passes as the environment of the new program
- Return value
  - Success: does not return
  - Error: -1

Running New Program

- `execve()` replaces the calling process with a new program



Running New Program: exec() Family

- execve(): execute a file

```
#include <unistd.h>
extern char **environ;

int execl(const char *pathname, const char *arg, ...
          /*, (char *) NULL */);
int execlp(const char *file, const char *arg, ...
          /*, (char *) NULL */);
int execle(const char *pathname, const char *arg, ...
          /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

List of arguments and array of pointers must be terminated by a null pointer

  - Replaces the current process image with a new process image
  - Above functions are based on `execve()`
  - *pathname*: path name that identifies the new process image file
  - `file`: the new process image file identified through directories in PATH environment variables

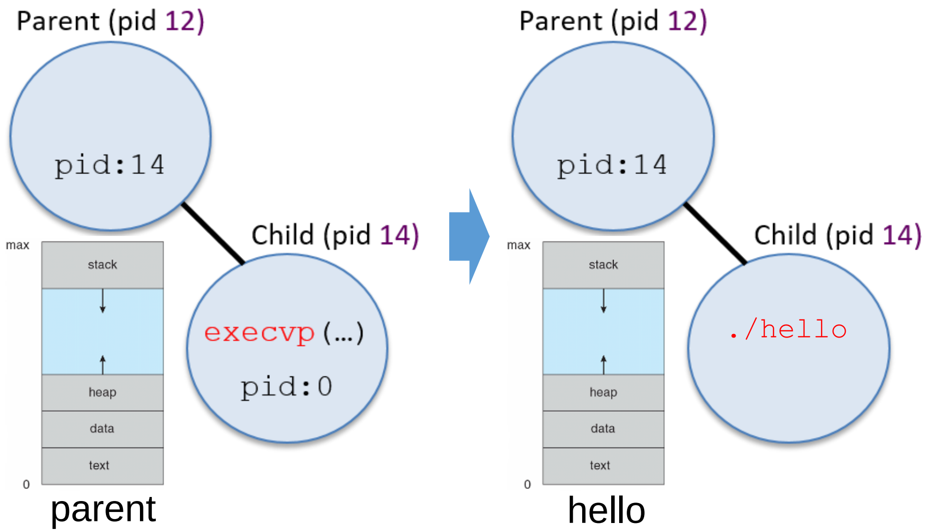
Running New Program: Example

exec.c

```
int main() {
    pid_t pid;
    int ret;
    char *argv[2];

    argv[0] = "./hello"; // initialize command line arguments for main
    argv[1] = NULL;

    pid = fork();
    if (pid == 0) {         // child process
        ret = execvp("./hello", argv);
        if (ret < 0) {
            perror("Error: execvp failed");
            exit(EXIT_FAILURE);
        }
    } else if (pid > 0) { // parent process
        wait(NULL);         // wait for the child process to complete
    } else {
        perror("Error: fork failed");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```




```
yunmin@yunmin:~/workspace/ch3$ gcc hello.c -o hello
yunmin@yunmin:~/workspace/ch3$ ls -l hello
-rwxrwxr-x 1 yunmin yunmin 15960 Mar 20 01:00 hello
yunmin@yunmin:~/workspace/ch3$ gcc hello.c -o hello
yunmin@yunmin:~/workspace/ch3$ ./exec
Hello, Handong!
```

Agenda

- Process Concept
- Process State and Scheduling
- Process API
  - Creating Process
  - Running New Program
  - Terminating and Waiting Process
- Signal
- Inter Process Communication

Terminating Process: exit()

- `exit()`: Cause normal process termination

```
#include <stdlib.h>
void exit(int status);
```

  - Terminates the current process
  - OS frees resources such as heap memory and open file descriptors and so on…
  - The least significant byte of status (i.e., status & 0xFF) is returned to the parent
    - Normally return with status 0
  - `atexit()` registers functions to be executed upon exit

Process Termination

- When process terminates, still consumes system resources
  - Various tables maintained by OS
- The parent process may wait for a child process to terminate in order to **reap** it
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- If no parent waiting, process is a **zombie**
  - Living corpse, half alive and half dead
- If parent terminated without waiting, process is an **orphan**

Process Termination

- What if Parent Doesn't Reap?
  - If any parent terminates without reaping a child, then child will be reaped by `init` process (`systemd`)
  - So, only need explicit reaping in long-running processes

  → shell, server, etc

Zombies: Example #1

zombie.c

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

int main()
{
    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 */
    }

    return 0;
}
```

```
yunmin@yunmin:~/workspace/ch3$ gcc zombie.c -o zombie
yunmin@yunmin:~/workspace/ch3$ ./zombie &
[1] 7844
yunmin@yunmin:~/workspace/ch3$ Running Parent, PID = 7844
Terminating Child, PID = 7846

yunmin@yunmin:~/workspace/ch3$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
yunmin      3306    2159  0 Mar19 pts/0    00:00:01 /bin/bash --init-fi
yunmin      7844    3306 91 01:25 pts/0    00:00:04 ./zombie
yunmin      7846    7844  0 01:25 pts/0    00:00:00 [zombie] <defunct>
yunmin      7891    3306 99 01:25 pts/0    00:00:00 ps -f
yunmin@yunmin:~/workspace/ch3$ kill 7844
yunmin@yunmin:~/workspace/ch3$ ps -f
[1]+  Terminated              ./zombie
UID          PID    PPID  C STIME TTY          TIME CMD
yunmin      3306    2159  0 Mar19 pts/0    00:00:01 /bin/bash --init-fi
yunmin      8031    3306 99 01:25 pts/0    00:00:00 ps -f
```

Zombie

Killing parent allows child to be reaped

Zombies: Example #2

nonterm.c

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

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

    return 0;
}
```

```
yunmin@yunmin:~/workspace/ch3$ gcc nonterm.c -o nonterm
yunmin@yunmin:~/workspace/ch3$ ./nonterm &
[1] 8290
Terminating Parent, PID = 8290
Running Child, PID = 8292
[1]+  Done                    ./nonterm
yunmin@yunmin:~/workspace/ch3$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
yunmin      3306    2159  0 Mar19 pts/0    00:00:01 /bin/bash --init-
yunmin      8292       1 98 01:32 pts/0    00:00:06 ./nonterm
yunmin      8318    3306 99 01:32 pts/0    00:00:00 ps -f
yunmin@yunmin:~/workspace/ch3$ kill -9 8292
yunmin@yunmin:~/workspace/ch3$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
yunmin      3306    2159  0 Mar19 pts/0    00:00:01 /bin/bash --init-
yunmin      8359    3306 99 01:32 pts/0    00:00:00 ps -f
```

- Child process still active even though parent has terminated.
- Child process must be killed explicitly, or else will keep running indefinetely

Waiting Process: `wait()`

- `wait()`: wait for process to change state

```
#include <sys/wait.h>
pid_t wait(int *wstatus);
```

  - Wait for state changes in a child of the calling process, and obtain information about the child whose state has changed
  - A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.
  - *`wstatus`*: If status is not NULL, `wait()` stores status information in the int to which it points. This integer can be inspected with the macros.
  - Return value
    - Success: child process ID
    - Error: -1
  - wait() is equivalent to waitpid(-1, &wstatus, 0);

https://man7.org/linux/man-pages/man2/wait.2.html

Waiting Process: waitpid()

- `waitpid()`: wait for process to change state

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Suspends execution of the calling process until a child specified by pid argument has changed state.
- *pid*: a child process ID that its parent process waits to stop
  - If pid is -1, the parent process waits for any child process to stop.
- *wstatus*: same with the status in `wait()`
- *options*
  - By default, `waitpid()` waits only for terminated children, but this behavior is modifiable via the options argument.
  - WNOHANG: return immediately if no child has exited
- Return value
  - Success: child process ID
  - Error: -1

Waiting Process: Macro Functions

- Macro function for handling returning status value from `wait()` and `waitpid()`

Macro function	Return value
WIFEXITED(wstatus)	returns true if the child terminated normally
WEXITSTATUS(wstatus)	returns the least significant 8 bits of the exit status that the child specified in if WIFEXITED returned true.
WIFSIGNALED(wstatus)	returns true if the child process was terminated by a signal (abnormal termination).
WTERMSIG(wstatus)	returns the number of the signal that caused the child process to terminate. This macro should be employed only if WIFSIGNALED returned true.

Waiting Process: Example #1

wait1.c

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

int main()
{
    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();
}
```

```
yunmin@yunmin:~/workspace/ch3$ gcc wait1.c -o wait1
yunmin@yunmin:~/workspace/ch3$ ./wait1
HP: hello from parent
HC: hello from child
Bye
CT: child has terminated
Bye
```

Waiting Process: Example #2

wait2.c

```c
#define N 5

int main()
{
    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 = 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);
    }
}
```

```
yunmin@yunmin:~/workspace/ch3$ gcc wait2.c -o wait2
yunmin@yunmin:~/workspace/ch3$ ./wait2
Child 8534 terminated with exit status 100
Child 8535 terminated with exit status 101
Child 8536 terminated with exit status 102
Child 8537 terminated with exit status 103
Child 8538 terminated with exit status 104
```

Waiting Process: Example #3

waitpid.c

```c
#define N 5

int main()
{
    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);
    }
}
```

```
yunmin@yunmin:~/workspace/ch3$ gcc waitpid.c -o waitpid
yunmin@yunmin:~/workspace/ch3$ ./waitpid
Child 8623 terminated with exit status 100
Child 8624 terminated with exit status 101
Child 8625 terminated with exit status 102
Child 8626 terminated with exit status 103
Child 8627 terminated with exit status 104
```

Waiting Process: Example #4

wait4.c

```c
char *argv[2];
argv[0] = "./hello";
argv[1] = NULL;

printf("A\n");

pid1 = fork();
if (pid1 == 0 ) {        /* child 1 */
    printf("B\n");

    pid2 = fork();
    if (pid2 == 0 ){     /* child 2 */
        printf("C\n");
        execvp("./hello", argv);
    } else {             /* child 1 (parent of child 2) */
        ret = wait(&status);
        printf("D\n");
        exit(0);
    }
} else {                 /* original parent */
    printf("E\n");
    ret = wait(&status);
    printf("F\n");
}
```

Expected results?

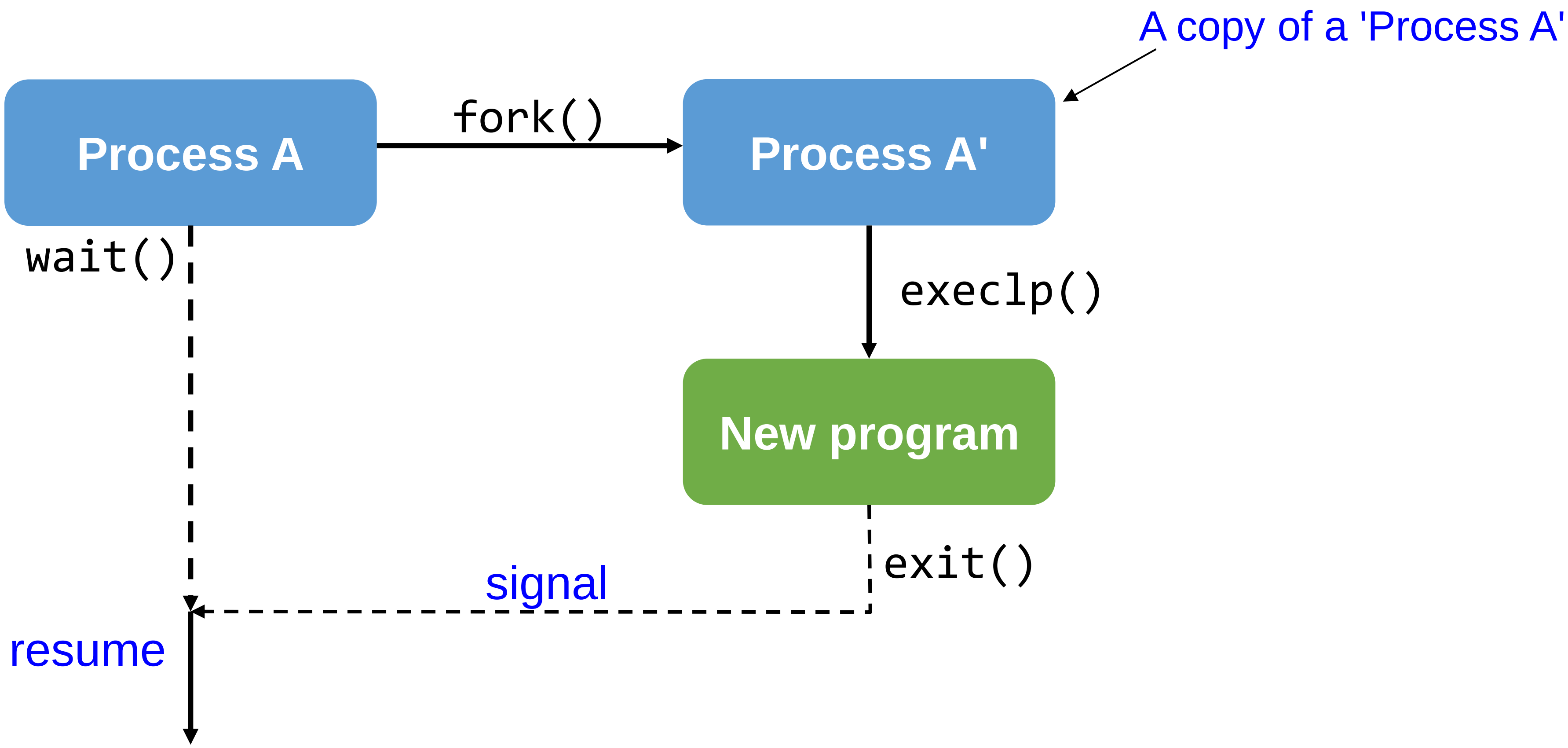
Summary: Executing Other Program

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

int main()
{
    pid_t pid = fork();                       // create a process
                                              // in general, pid_t is defined as int
    if (pid < 0) {                            // error occurred
        fprintf(stderr, "fork failed\n");
        return 1;
    } else if (pid == 0) {                    // child process
        execlp("/bin/ls", "ls", NULL);
    } else {                                  // parent process
        wait(NULL);                           // waits for child process to complete
        printf("Child Completed\n");
    }
    return 0;
}
```

HANDONG GLOBAL UNIVERSITY

Summary: Executing Other Program



Summary: Executing Other Program