CS 149 Operating Systems
Processes

Instructor: Kong Li

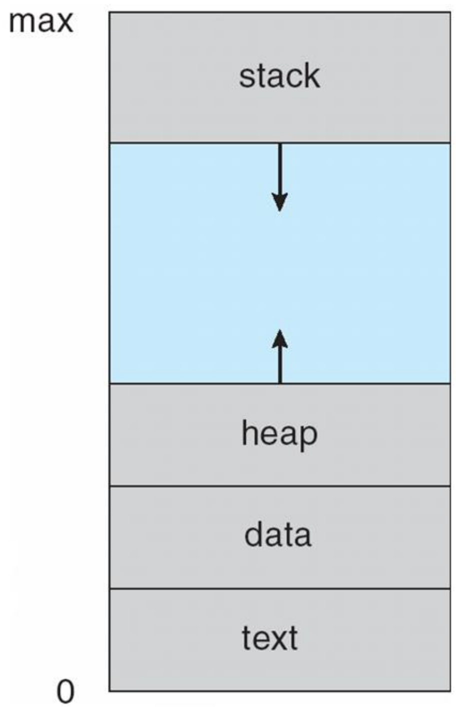
Content

- Process Concept
- Context switch
- Process Scheduling
- Operations on Processes
  - fork, exec, wait, exit
- Interprocess Communication (IPC)
  - Shared memory, msg passing, socket, pipe

Process Concept

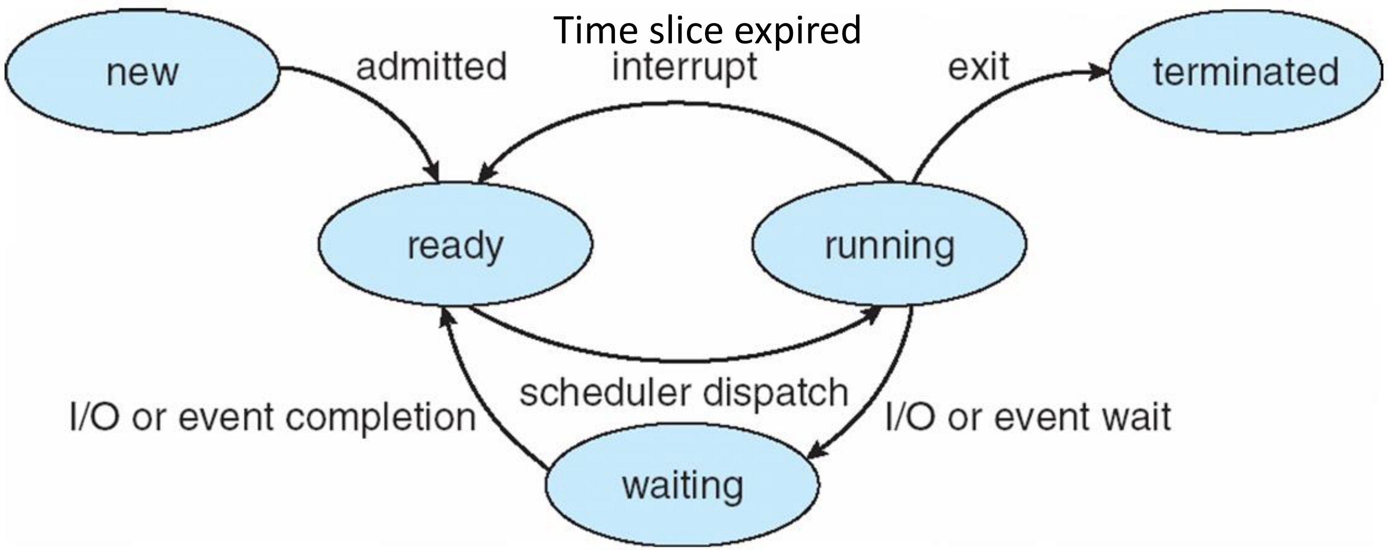
- Program: passive entity stored on disk (executable file)
  - Program becomes proc when executable file loaded into memory
  - One single program can become several concurrent procs
- Process: active entity in memory, a program in execution
  - proc exec: in sequential fashion
- Process parts
  - program counter, CPU registers, proc state (later)
  - Text: program code
  - Data: global variables
  - Heap: runtime memory allocation (malloc/free)
  - Stack: temporary data
    - Function parameters, return addr, local variables

UNIX proc layout



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

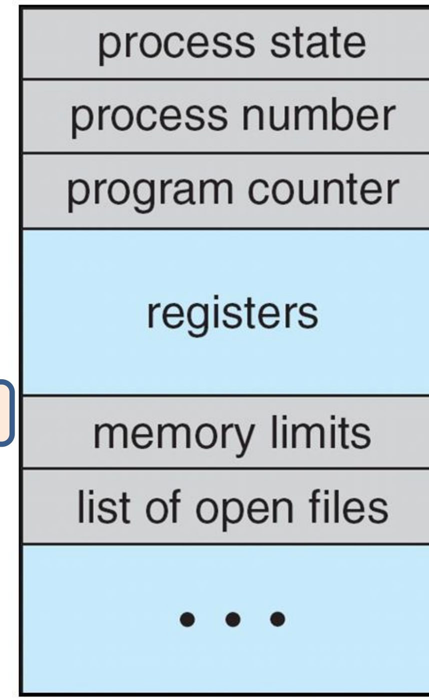


State transition diagram

Process Control Block (PCB)

- PCB: one per proc (aka task control block)
  - Proc state: running, waiting, etc
  - PID: unique among all procs on a given computer
  - Program counter: next instruction to execute
  - Registers: contents of CPU registers

    include SP
  - scheduling: priorities, queue pointers
  - Memory-mgmt: memory allocated to proc
  - Accounting: CPU used, clock time elapsed since start, time limits
  - I/O status: – I/O devices allocated to proc, open files



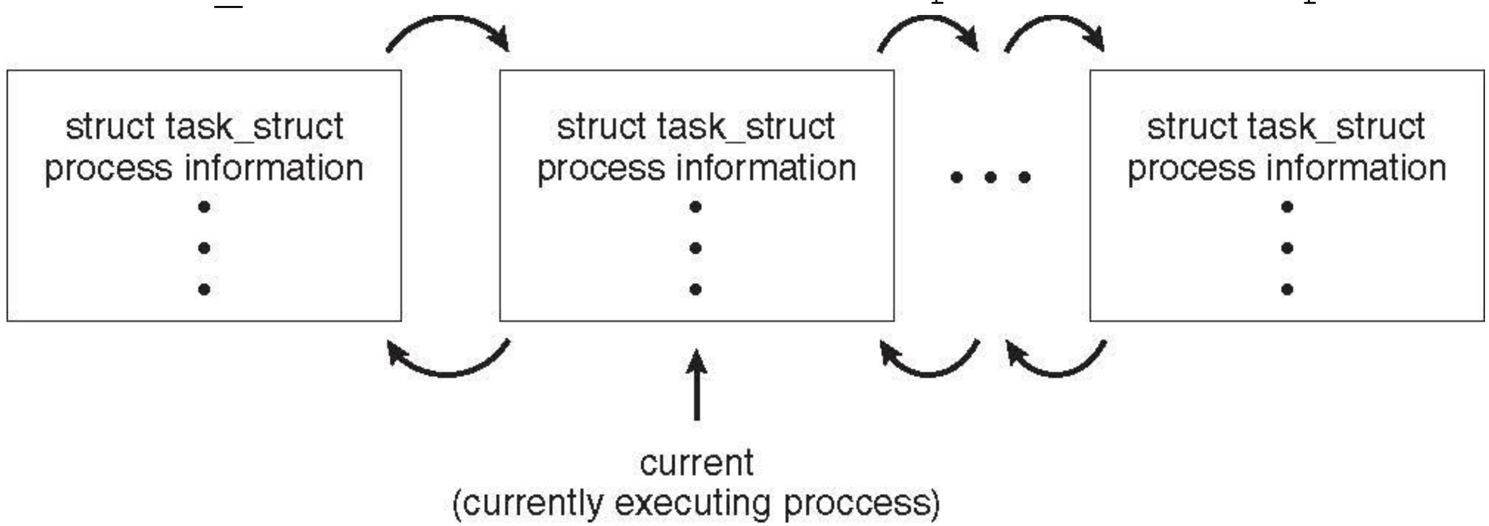
Q: Where is PCB?
A: kernel memory
(in main memory)

Process Representation in Linux

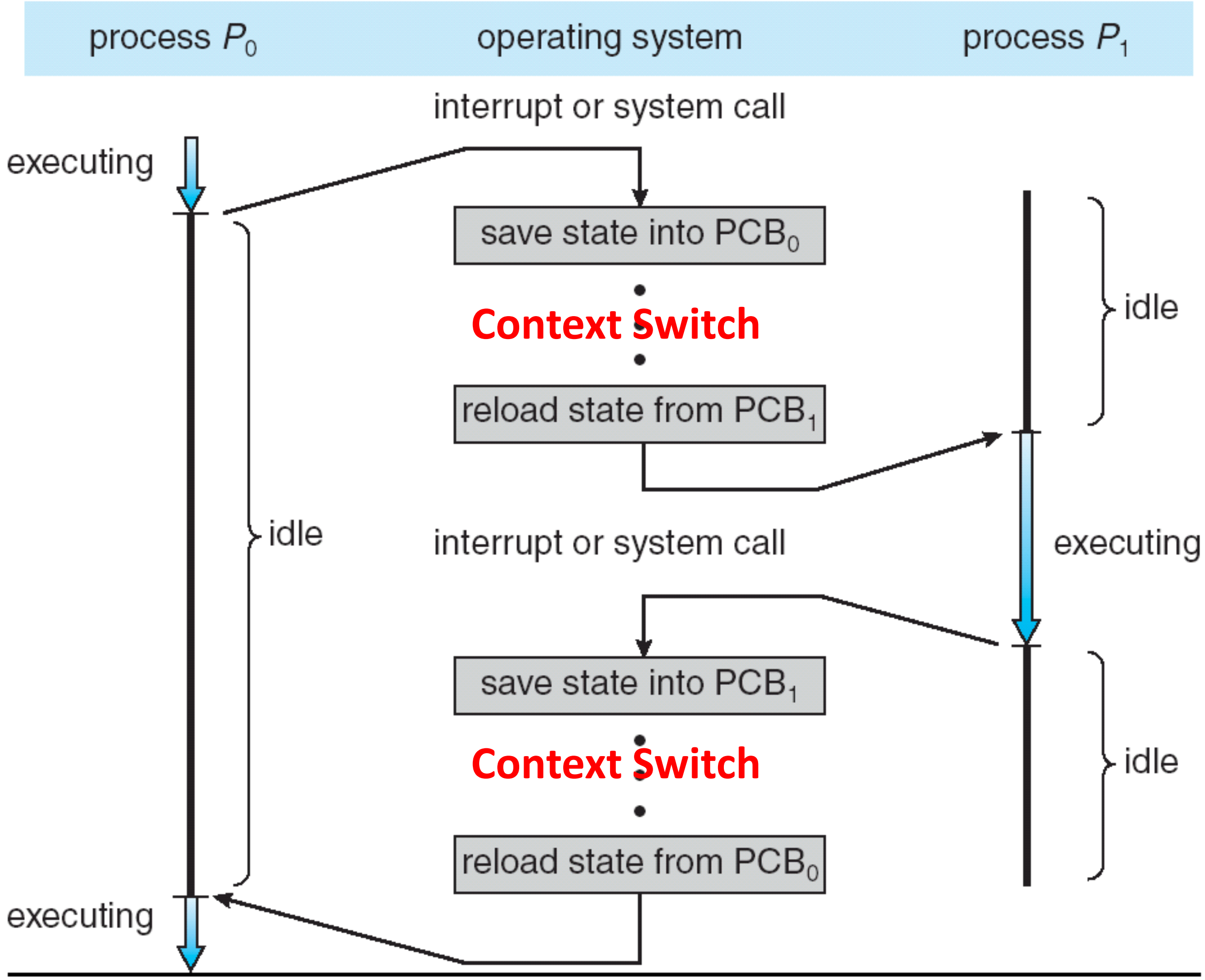
- Represented by the C structure `task_struct`

```
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 proc */
```

Why?



CPU Switch From Process to Process

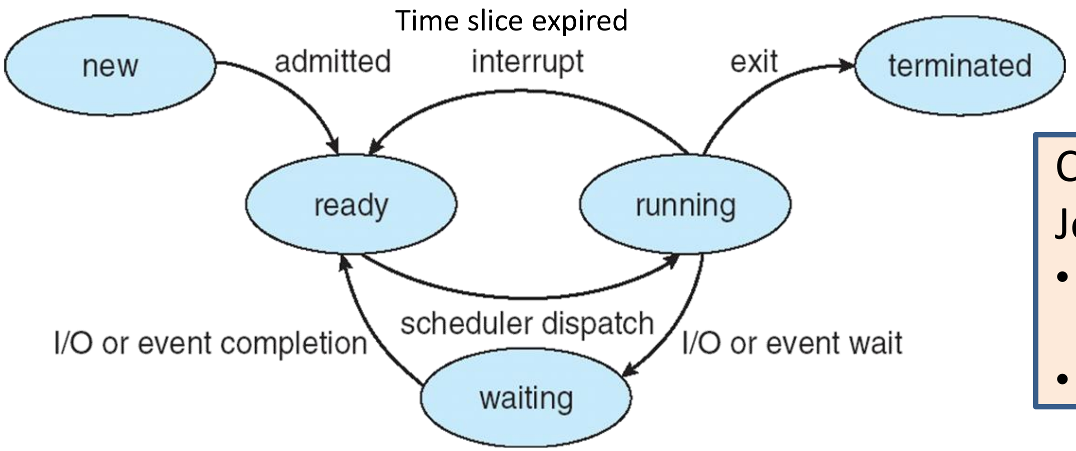


Context Switch

- **Context** of a proc represented in the PCB
- context switch: CPU switches from one to another proc
  - OS must save the state of the old proc
  - Then OS must load (restore) the saved state of the new proc
- Context-switch time is overhead
  - the system does no useful work while switching
  - complicated OS/PCB → longer context switch
  - HW support: multiple sets of registers? Etc.
- How to improve context switch efficiency
  - Reduce the frequency of context switch
  - Speed up context switch: multiple set of HW registers, etc.

Process Scheduling

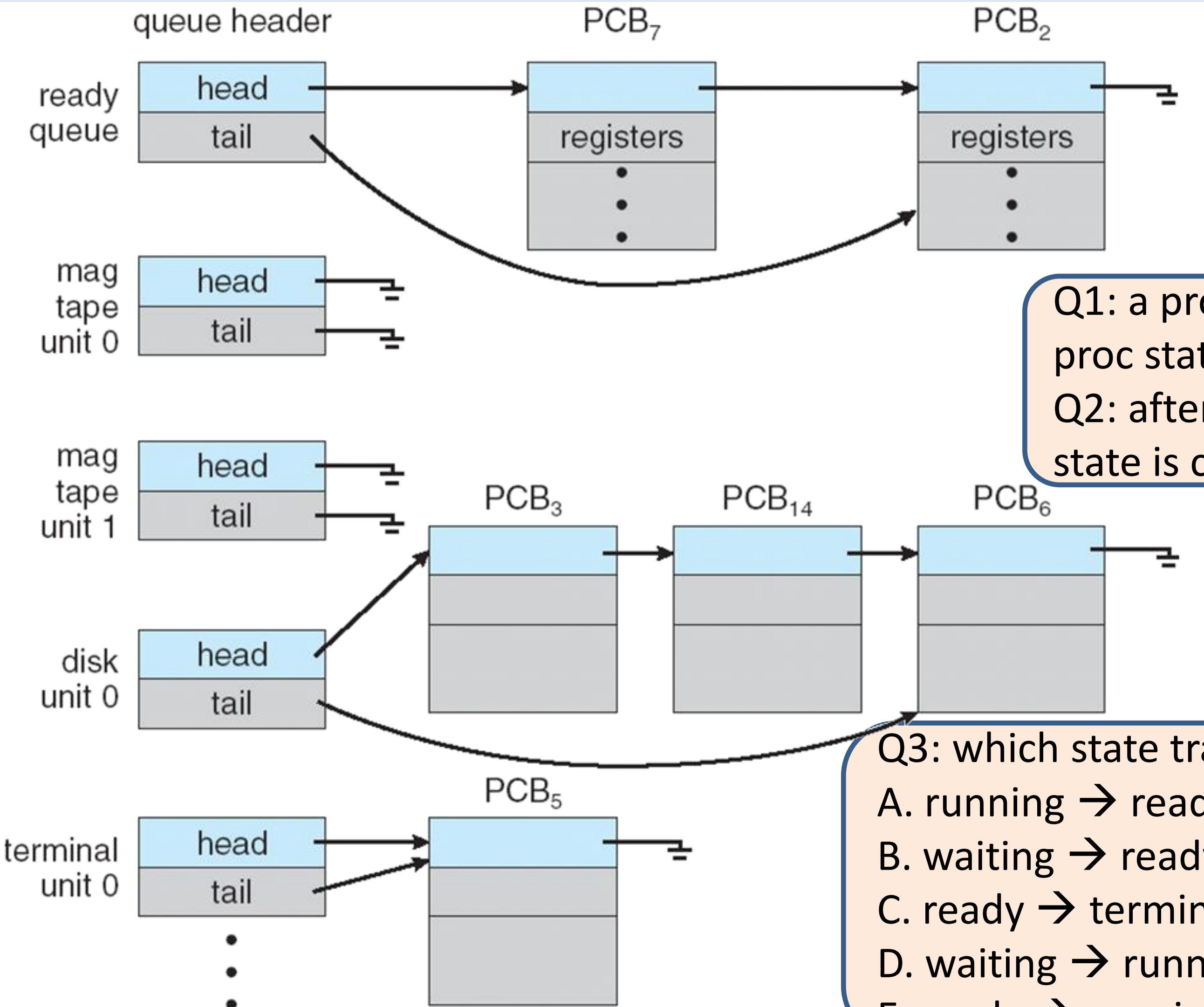
- Goals: maximize CPU use, quickly switch procs onto CPU for time sharing
- Various scheduling queues:
  - **Ready Q**: procs that are ready and waiting to execute
  - **Device Qs**: set of procs waiting for I/O devices (one Q per device)
- Procs migrate among various Qs, based on proc state transition
  - continues this until terminates (removed from all Qs & deallocated resources)



CPU scheduler: runs frequently
Job scheduler: runs less frequently
- Select a good mix of CPU-bound and I/O-bound procs
- Not exist on UNIX & Windows

- CPU (short-term) scheduler: selects among ready procs for execution
- Job (long-term) scheduler: controls degree of multiporogramming

Ready Queue, I/O Device Queues

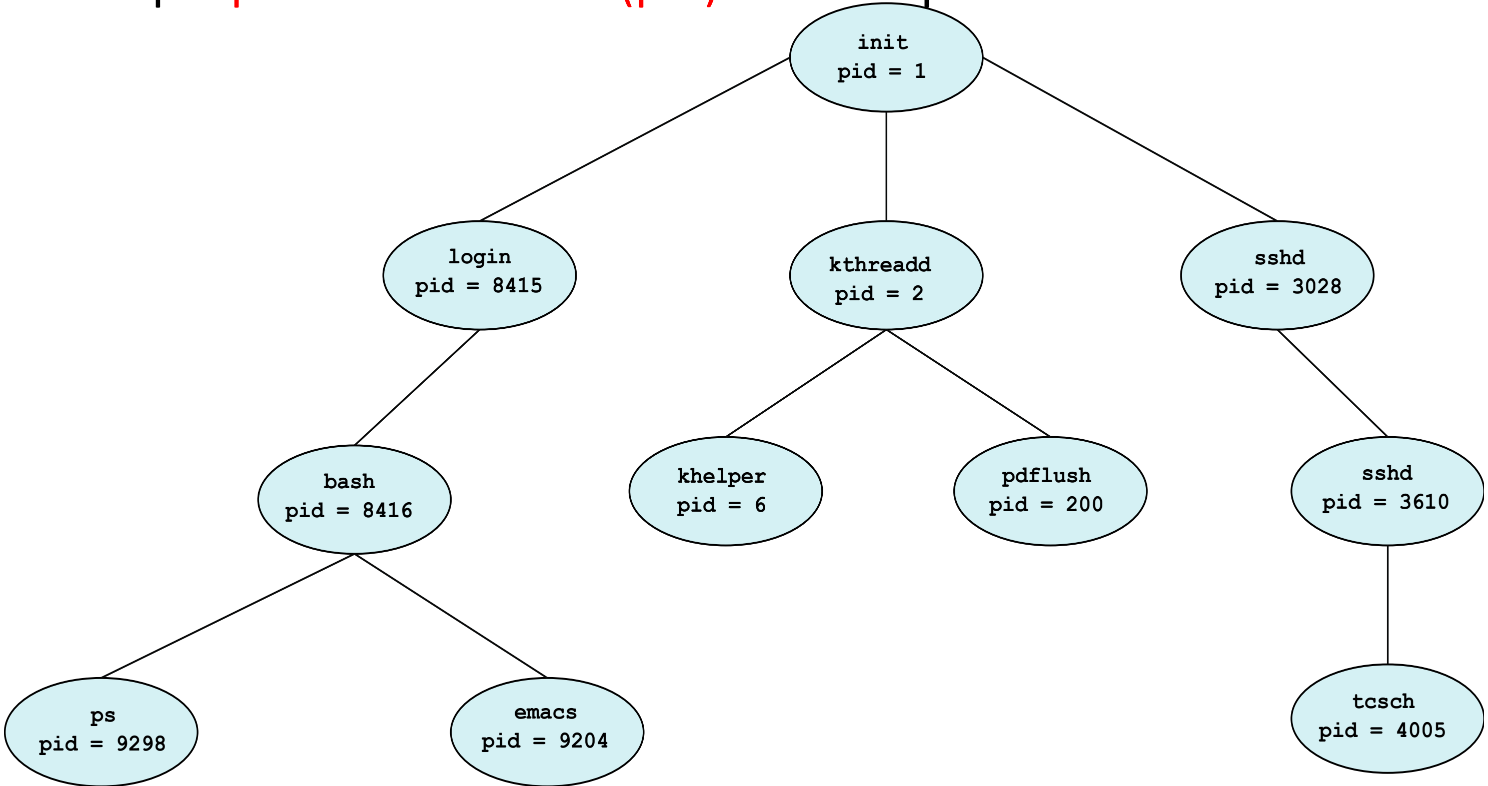


Q1: a proc invokes sleep(10), the proc state is changed from ? to ?
Q2: after 10 seconds, the proc state is changed from ? to ?

Q3: which state transition are not valid?
A. running → ready
B. waiting → ready
C. ready → terminated
D. waiting → running
E. ready → running

Process Tree/Hierarchy

- Linux/UNIX process tree
  - Parent proc creates children procs
  - Child proc creates grandchild procs, etc.
  - PCB records parent proc and children procs
  - unique process identifier (pid) for each proc

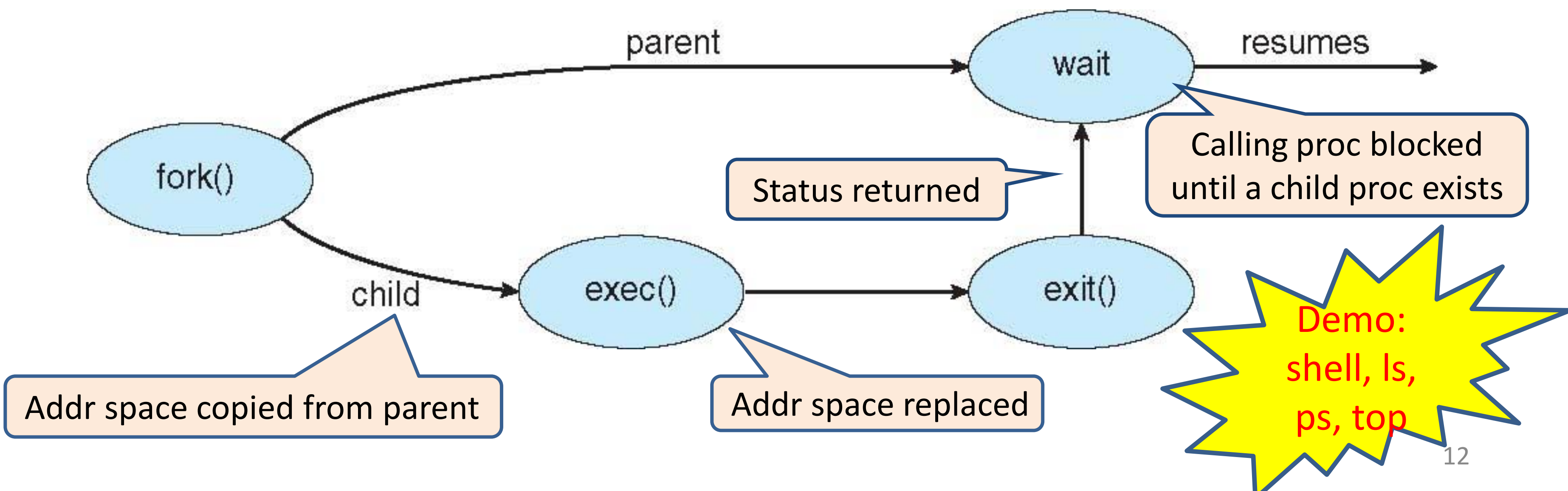


Process Creation

- UNIX system calls
  - **`fork()`**: creates new proc – dup of parent proc
    - Both parent and child procs are ready/running
  - **`exec()`**: replace the proc's memory space w/ a new program
    - Usually called after a `fork()`
    - Ex: shell (bash, csh, etc)

man fork
man exec

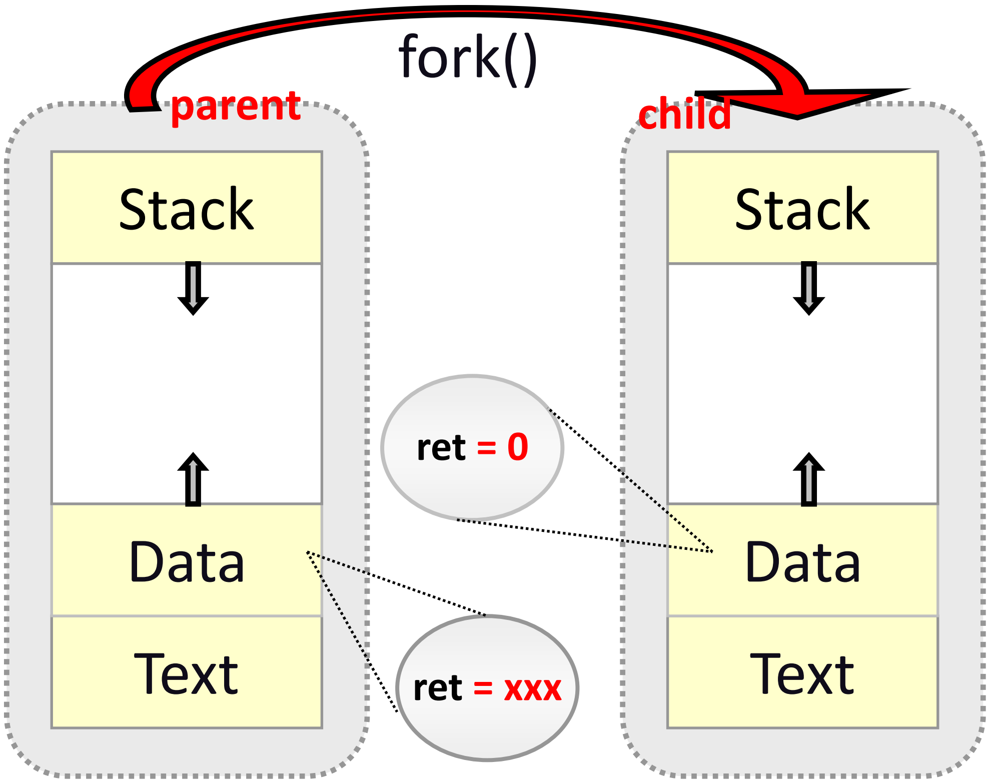
- Unlike Windows, no UNIX sys call creates a different new proc



fork()

```
#include <unistd.h>

pid_t fork(void);
```

man fork



- Current parent process is cloned to a new child proc
  - Both parent and child procs are ready/running
  - Separate memory space
  - The child proc inherits memory (text, data, heap, stack) & "open files" from parent (except?)
  - Both procs return from `fork()`

```
ret = fork();
```

`ret` == -1 if unsuccessful

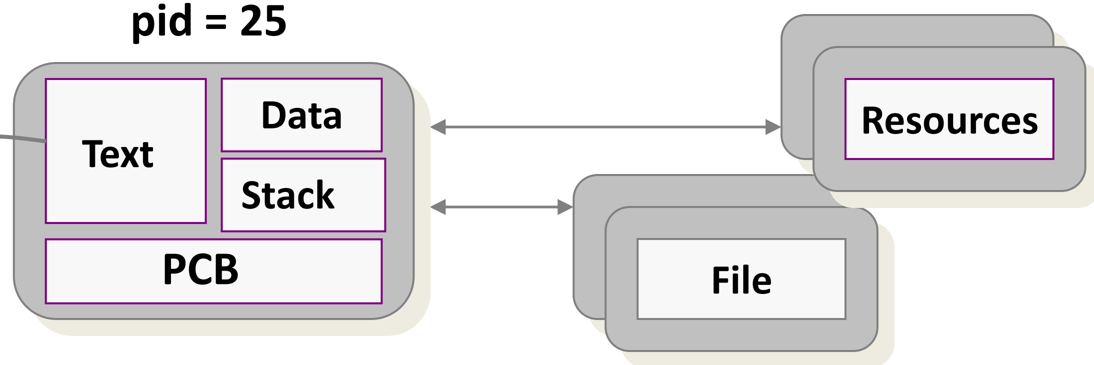
`ret` == 0 in the child

`ret` == child's PID in the parent

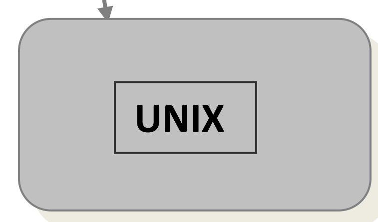
- adapted from Prof. Thomas Way thomas.way@villanova.edu

How fork Works (1)

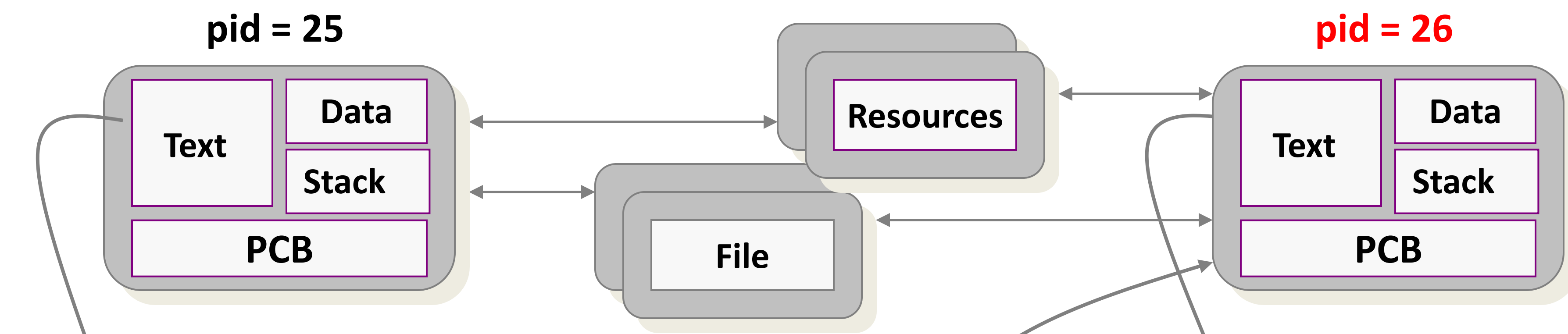
pid = 25




```
ret = fork();
switch(ret)
{
   case -1:
               perror("fork");
               exit(1);
    case 0:  // I am the child
             <code for child >
               exit(0);
    default:  // I am parent ...
               <code for parent >
               wait(0);
}
```

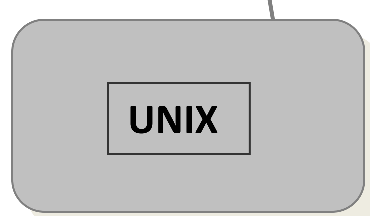


How fork Works (2)



ret = 26

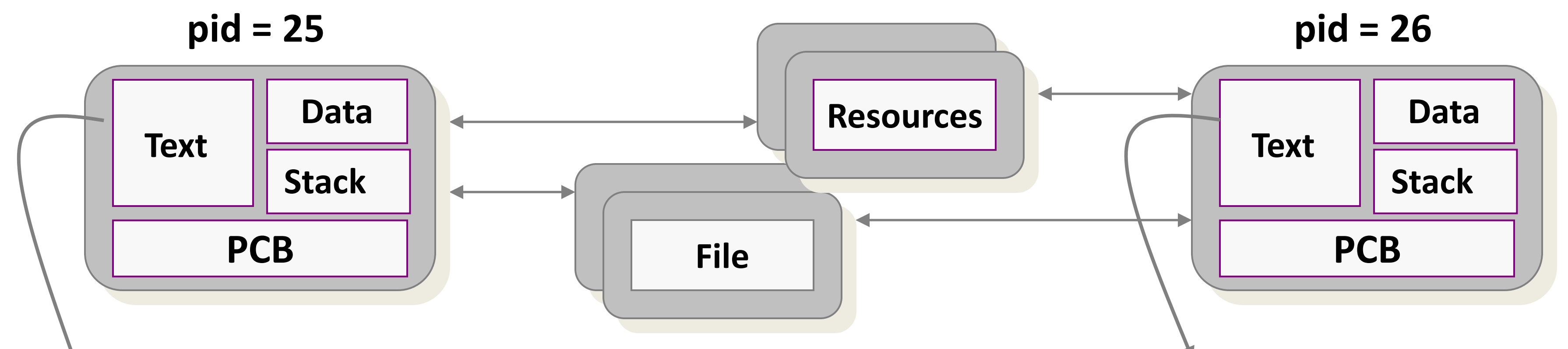
```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```



ret = 0

```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

How fork Works (3)

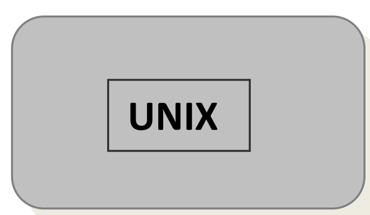


ret = 26

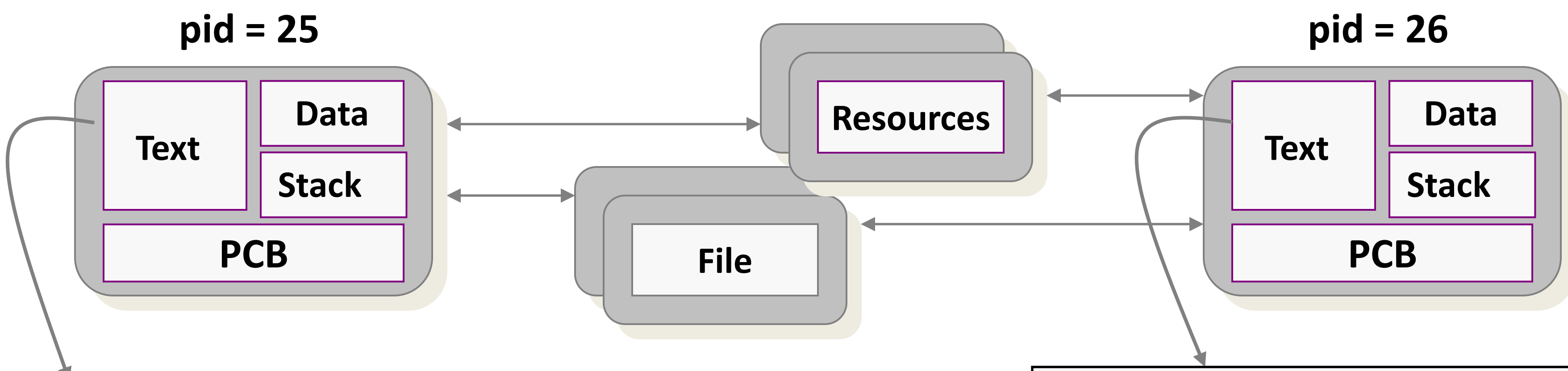
```
ret = fork();
switch(ret)
{
  case -1:
          perror("fork");
          exit(1);
  case 0:  // I am the child
          <code for child >
          exit(0);
  default:  // I am parent ...
          <code for parent >
          wait(0);
}
```

ret = 0

```
ret = fork();
switch(ret)
{
  case -1:
          perror("fork");
          exit(1);
  case 0:  // I am the child
          <code for child >
          exit(0);
  default:  // I am parent ...
          <code for parent >
          wait(0);
}
```

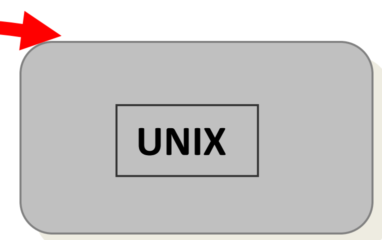


How fork Works (4)



ret = 26

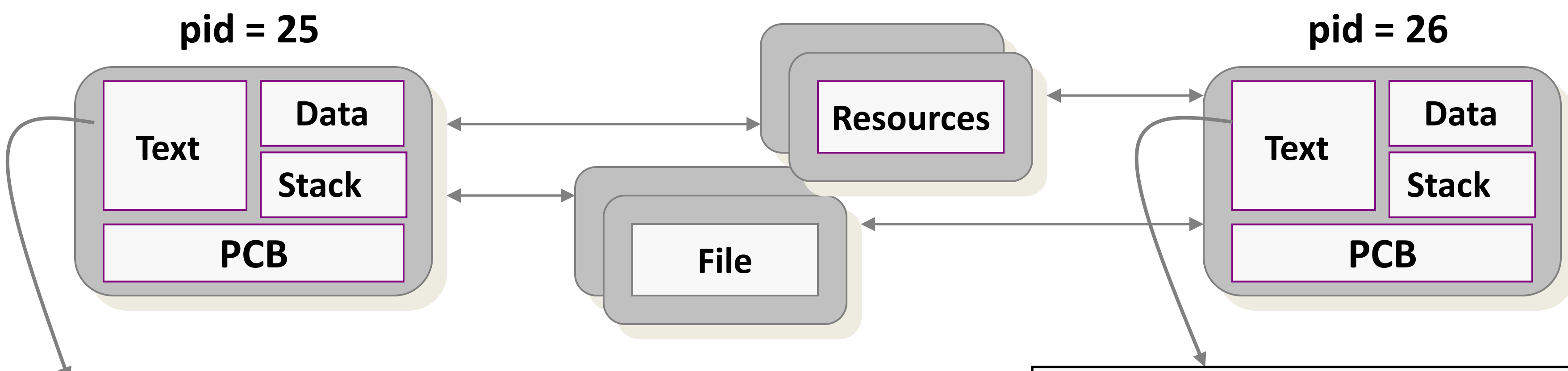
```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default: // I am parent ...
            <code for parent >
            wait(0);
}
```



ret = 0

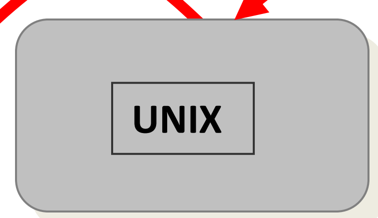
```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default: // I am parent ...
            <code for parent >
            wait(0);
}
```

How fork Works (5)



ret = 26

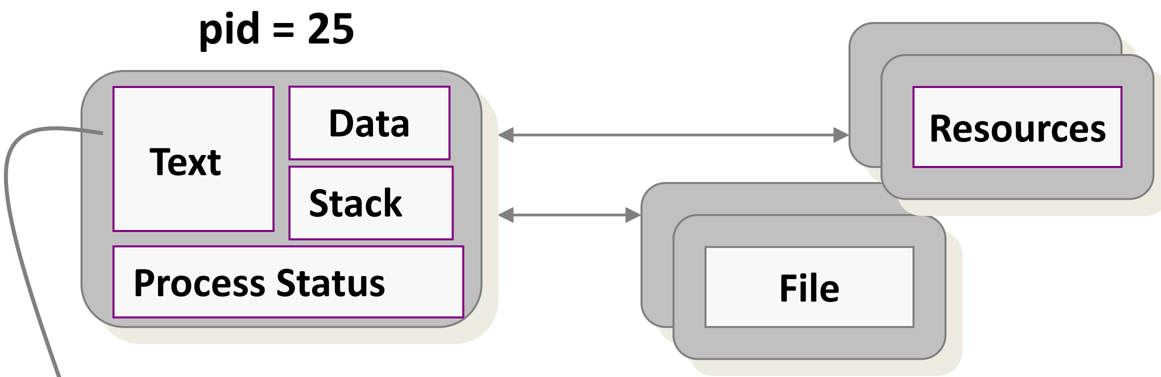
```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```



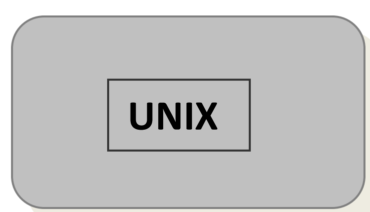
ret = 0

```
ret = fork();
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
            <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
}
```

How fork Works (6)




```
ret = fork();                ret = 26
switch(ret)
{
   case -1:
            perror("fork");
            exit(1);
   case 0:  // I am the child
          <code for child >
            exit(0);
   default:  // I am parent ...
            <code for parent >
            wait(0);
            < ... >
```



Example: fork

```
#include <stdio.h>
int num = 0;

int main(){
    int pid;
    pid = fork();

    printf("%d", num);
    if(pid == 0) {        /*child*/
        num = 1;

    } else if(pid > 0) {   /*parent*/
        num = 2;
    }
    printf("%d", num);
}
```

parent: 0

child: 0

What if the child calls fork() here?

parent: 2

child: 1

exit()

```
#include <stdlib.h>

void exit(int status);
```

man exit

- **`status`**: child's exit code (lower 8-bit only) returned to its parent
- Normal termination: closes all files, deallocates resources (memory, I/O buffer, etc.), removes child zombie procs, if any.
- Proc's PCB may or may not be released immediately
- If parent proc is blocked in `wait()`, unblock the parent proc & release child's PCB
- If parent proc hasn't called `wait()`, holds the exit code in PCB until the parent calls `wait()`
  – the child proc does not really die, but it enters a zombie/defunct state

wait()

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

man wait

- Parent may want to wait for children to finish
  - Ex: a shell waiting for operations to complete
- **`wait()`**: wait for any child to terminate
  - Blocks until some child terminates
  - Returns the process ID of the child proc, and child's exit status
  - Or returns -1 if no children exist (i.e., already exited)
- **`waitpid()`**: wait for a specific child to terminate
  - Blocks till a child with particular proc ID terminates
- Both returns pid of the child proc that was terminated

What about wait(0);

Example: exit, wait

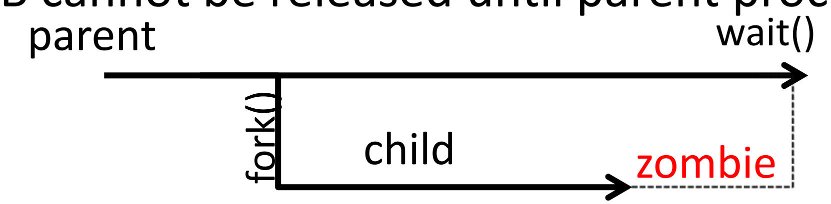
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
 pid_t fpid, pid, wpid;

 /* fork a child process */
 fpid = fork();
 if (fpid < 0) {
  /* error occurred */
  fprintf(stderr, "Fork Failed");
  return 1;
 }
 else if (fpid == 0) { /* child process */
   pid = getpid();
   printf("child: fpid = %d\n", fpid);
   printf("child: pid = %d\n", pid);
   exit(23);
 } else { /* parent process */
   int status = 0;
   pid = getpid();
   printf("parent: fpid = %d\n", fpid);
   printf("parent: pid = %d\n", pid);

   wpid = wait(&status);
   printf("parent: wait: wpid = %d, status = %d,
exit code = %d\n", wpid, status,
WEXITSTATUS(status));
 }
 return 0;
}
```

Process Termination

- **`exit()`**: normal proc termination
- **`abort()`**: abnormal termination
- **`kill()`**: terminate another proc
  - Permission: privileged user, or w/ the same user id

man abort
man 2 kill
man 1 kill

- **zombie process** (or defunct process): a proc has terminated, but its parent proc has not yet called `wait()`
  - PCB cannot be released until parent proc invokes **`wait()`**



How to remove zombie process?

- **orphan process**: If a parent terminated w/o invoking `wait()`, child procs become orphan
  - OS assigns a system proc as the new parent
  - The new parent proc invokes `wait()` from time to time (why?)

Unix: init (pid 1)
Linux: init (pid may or may not be 1), upstart, etc.

exec()

```
#include <unistd.h>

int execv(const char * path, char * const argv[]);
```

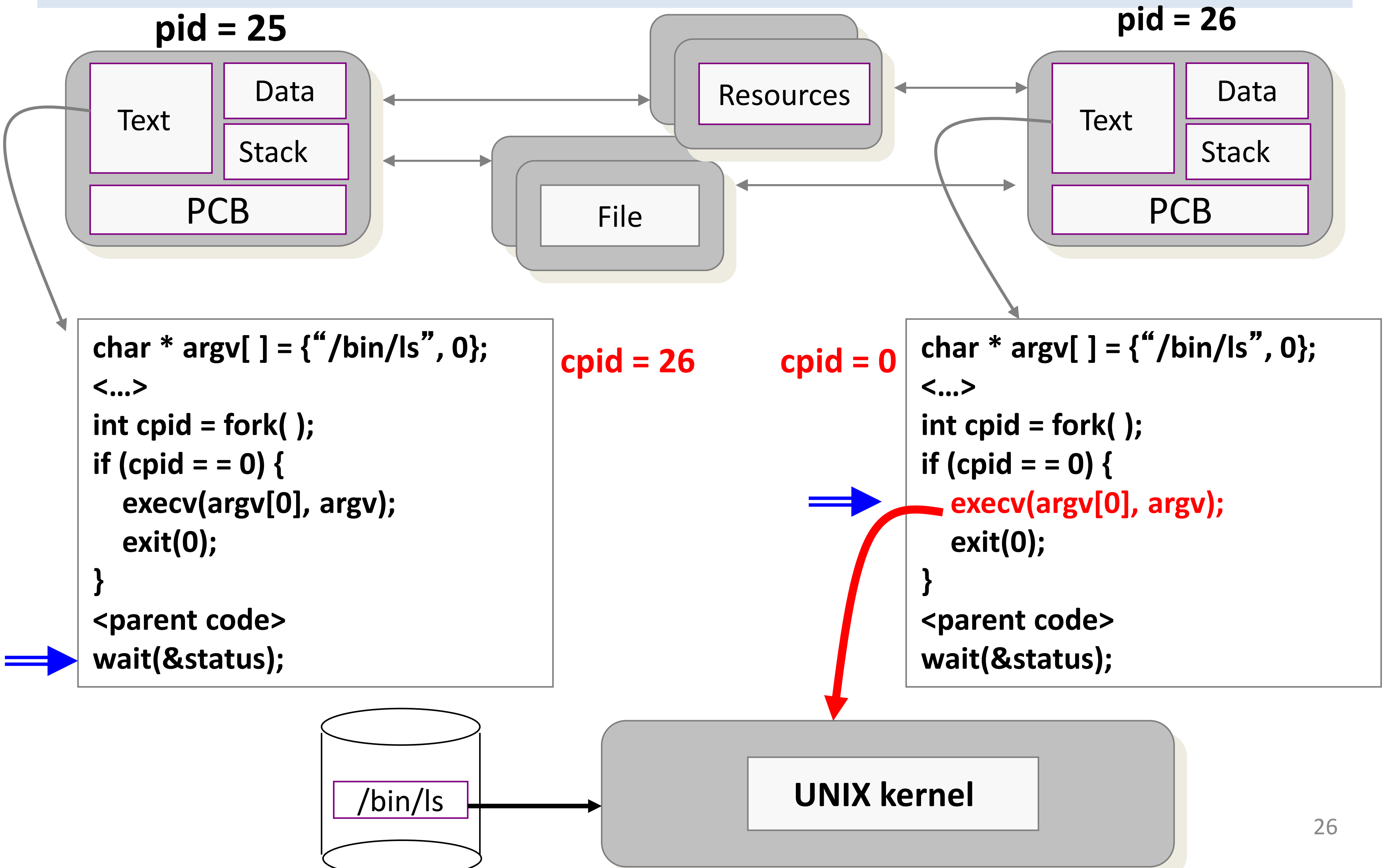
man 3 exec

- executes a program - replacing the calling proc with a new proc
  - path: full path for the program to be executed
  - argv: the array of arguments for the program to execute
    - each argument is a null-terminated string
    - the first argument is the name of the program
    - the last entry in argv is NULL

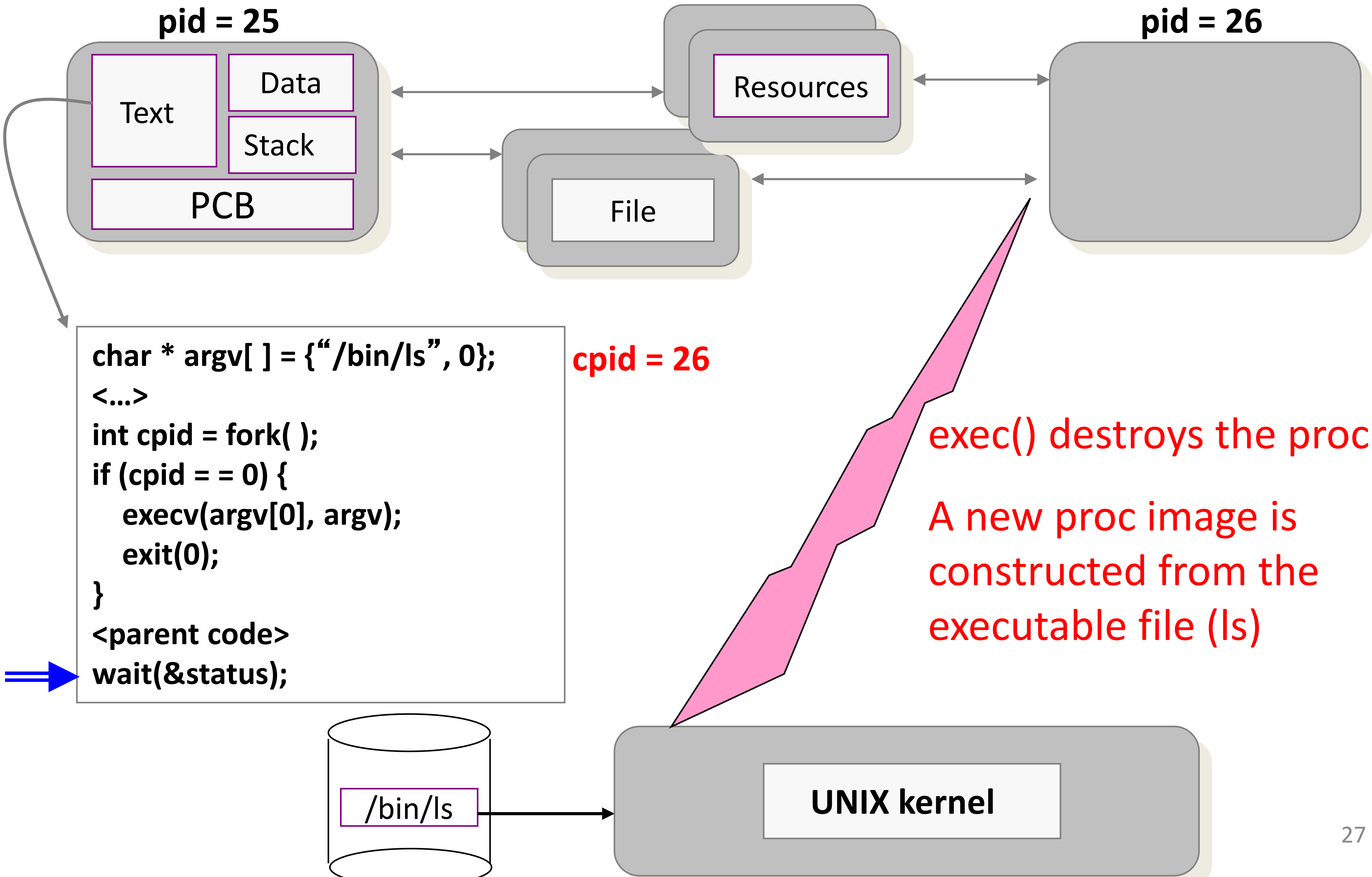
Why?

- After a successful exec, no return to the calling proc
  - calling proc replaced by the new proc
  - The new proc has the same pid (and parent pid) as the calling proc
- Return -1 only when error (calling proc still alive)

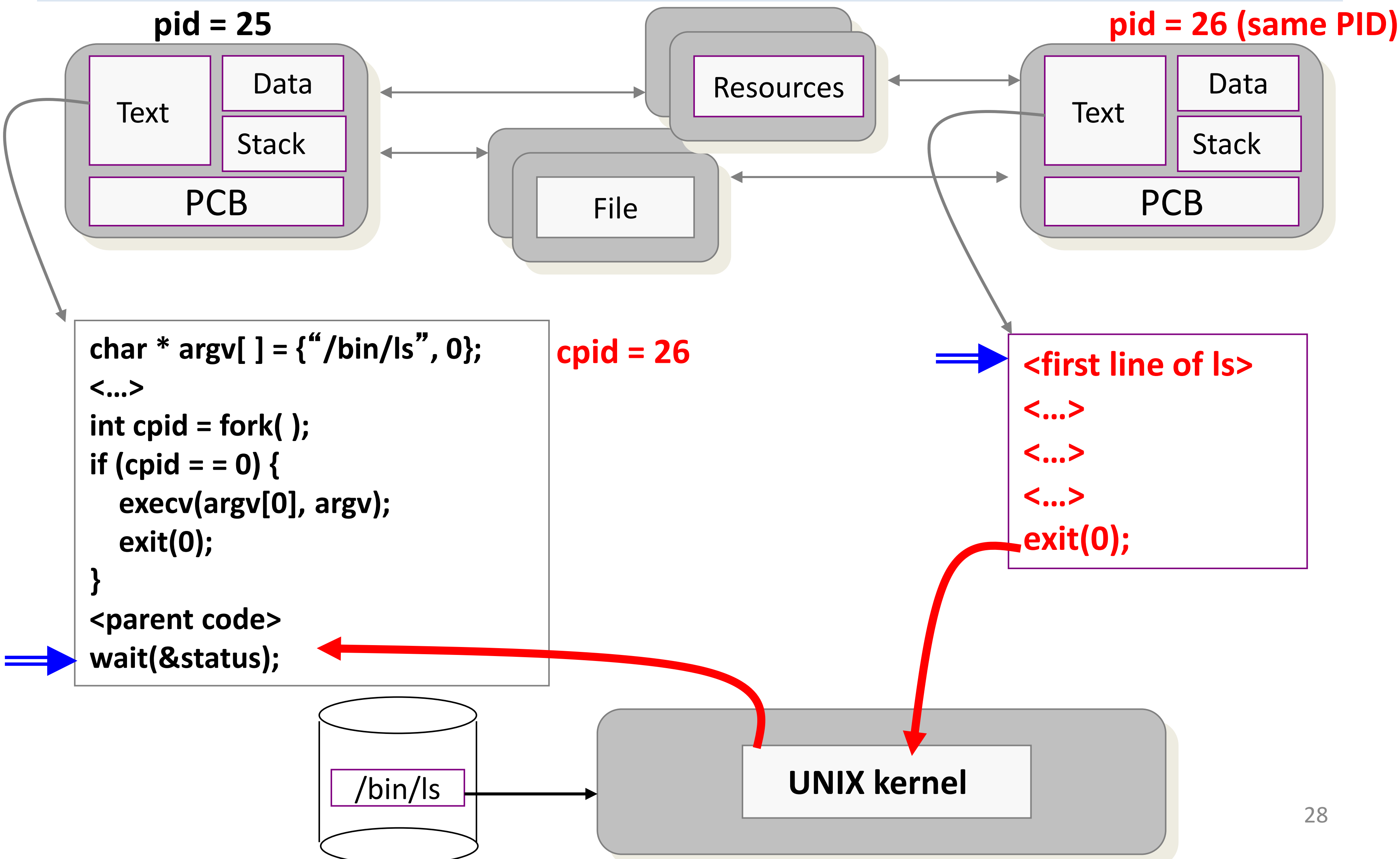
How execv Works (1)



How execv Works (2)



How execv Works (3)



execv Example

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

char * argv[] = {"/bin/ls", "-l", 0};
int main()
{
   int pid, status;

   if ( (pid = fork() ) < 0 ) {
        printf("Fork error \n");
        exit(1);
   }
   if(pid == 0) { /* Child executes here */
        execv(argv[0], argv);
        printf(“Exec error \n”);
        exit(1);
   } else {     /* Parent executes here */
        wait(&status);
   }
   printf("Hello there! \n");
   return 0;
}
```

```
argv[0] = "/bin/ls";
argv[1] = "-l";
argv[2] = NULL;
```

Note the NULL string at the end

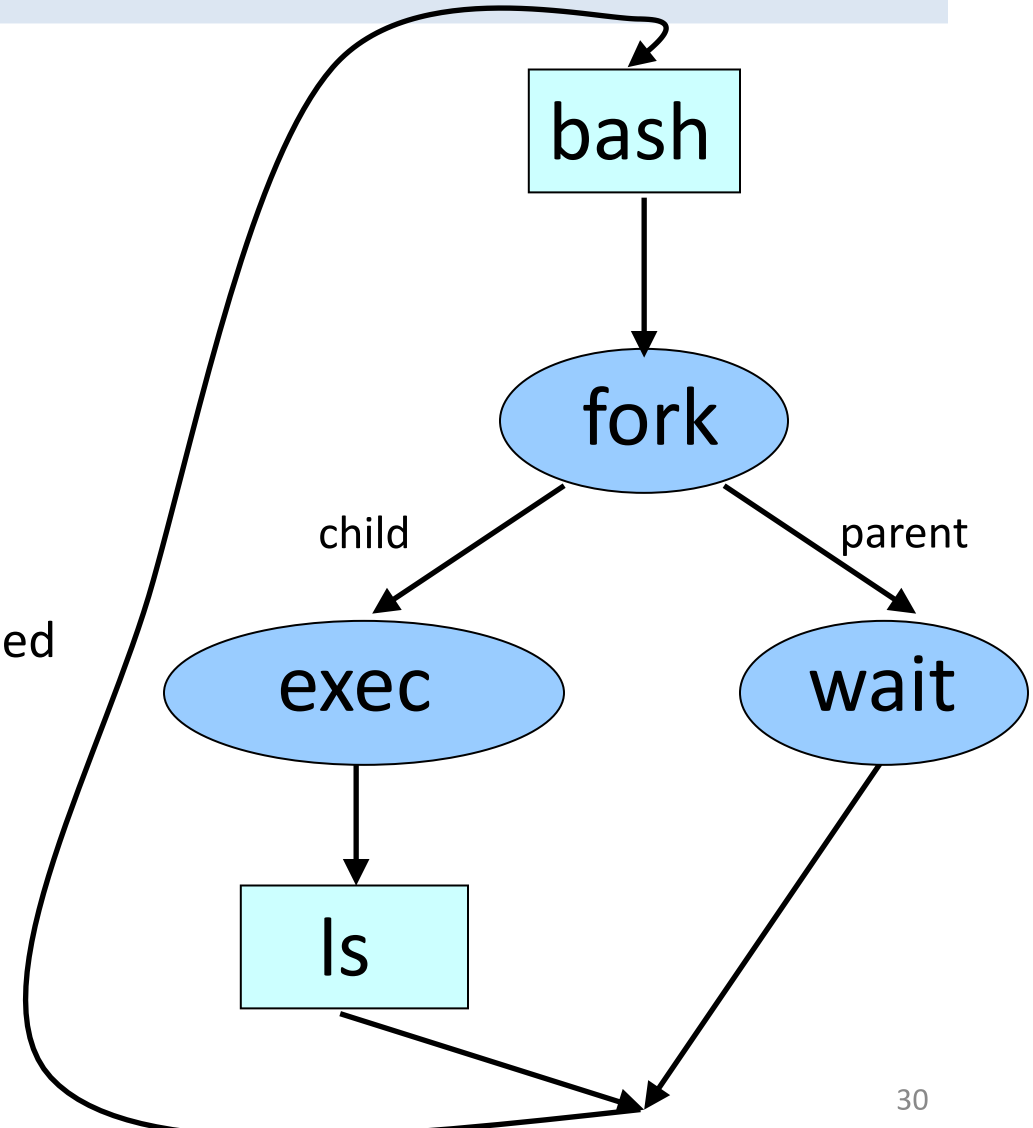
Error handling

Error handling

Demo!

Example: A Simple Shell

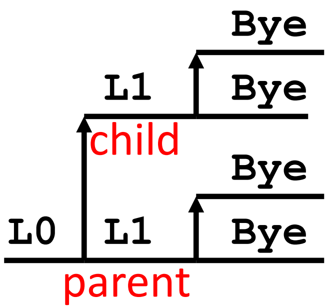
- Shell is the parent process
  - E.g., bash
- Reads & parses cmd line
  - E.g., “`ls -l`”
- Invokes child proc
  - `fork`, `exec`
- Waits for child
  - `wait`
- Each Linux proc has three opened streams
  - stdin: standard input
    - keyboard input → stdin
  - stdout: standard output
    - printf → stdout
  - stderr: standard error



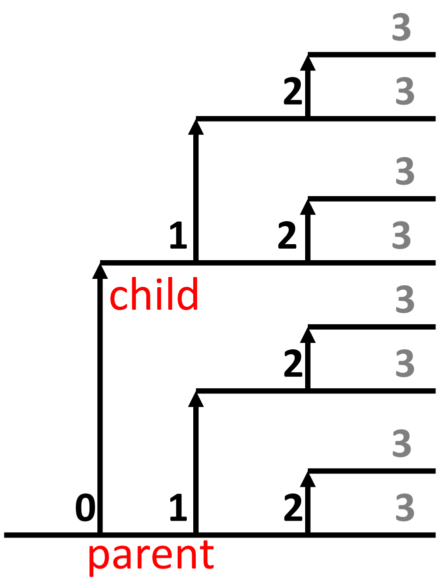
Fork Example 2 & 3: Spacetime Diagram

- Key Point: keep track of the execution of each process

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




```
void fork3()
{ int i;
  for (i=0; i<3; i++) {
    fork();
  }
}
```



Fork Example 4 & 5: Spacetime Diagram

- How many processes? `L0`? `L1`? `L2`?
- The value of `i` in each process in spacetime diagram?

```
void fork4()
{   int i = 4;
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
        i = i + 2;
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    ++i;
    printf("Bye\n");
}
```

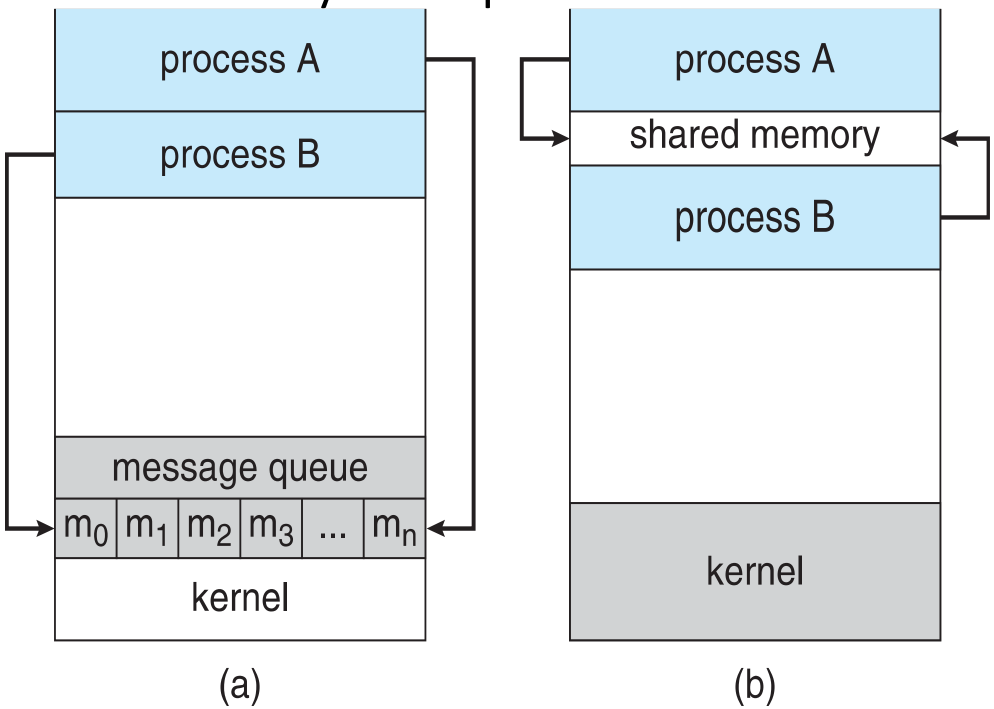
```
void fork5()
{   int i = 5;
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
        --i;
       if (fork() == 0) {
           printf("L2\n");
           fork();
            ++i;
       }
    }
    printf("Bye\n");
}
```

Multiprocess Architecture – Chrome Browser

- Many web browsers as single proc
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser: multiproc w/ 3 types of procs:
  - Browser proc: manages UI, disk and network I/O
  - Renderer proc: 1 per website - render web pages HTML, Javascript
  - Plug-in proc: for each type of plug-in
  - Pros? Cons?

Interprocess Communication (IPC)

- Procs within a system may be independent or cooperating
  - Independent proc: cannot affect or be affected by other running proc
  - Cooperating proc: can affect or be affected by other procs
    - Info sharing
    - Computation speedup
- Cooperating procs need IPC
- Two models of IPC
  - Message passing: (a)
    - same or different computer
    - good for smaller amount of data exchange
  - Shared memory: (b)
    - same computer only
    - faster & more efficient for large data exchange
    - need synchronization & overhead (later)



IPC - Shared-Memory: bounded buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;   /* idx – producer */
int out = 0;  /* idx – consumer */
```

- can only use BUFFER_SIZE-1 elements

Consumer

```
item next_consumed;
while (true) {
  while (in == out);  /* do nothing */
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;

  /* consume the item in next_consumed */
}
```

Producer

```
item next_produced;
while (true) {
  /* produce an item in next_produced */
  while (((in + 1) % BUFFER_SIZE) == out);/* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Problems?

IPC - POSIX Shared Memory

Producer

gcc .... -lrt

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

Conumser

gcc .... -lrt

man shm_open
man mmap
man ftruncate

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Demo!

parent/child proc ?

Problems?

Sync & Async Message Passing

- Msg passing: a form of synchronization
- Msg passing: blocking or non-blocking
- Blocking ≡ synchronous
  - **Blocking send** -- the sender is blocked until msg is received
  - **Blocking receive** -- the receiver is blocked until msg is available
  - Involves proc state transition
- Non-blocking ≡ asynchronous
  - **Non-blocking send** -- the sender sends msg and continue
  - **Non-blocking receive** -- the receiver receives a valid msg or null msg
  - Why useful?
  - Do not have to involve proc state transition (why?)
- Different combinations possible
- I/O operations in OS: synchronous, asynchronous

Rendezvous

- Rendezvous: both send and receive are blocking
- Producer-consumer becomes trivial

Producer

```
message next_produced;
while (true) {
   /* produce an item in next_produced */
   send(next_produced);
}
```

Consumer

```
message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next_consumed */
}
```

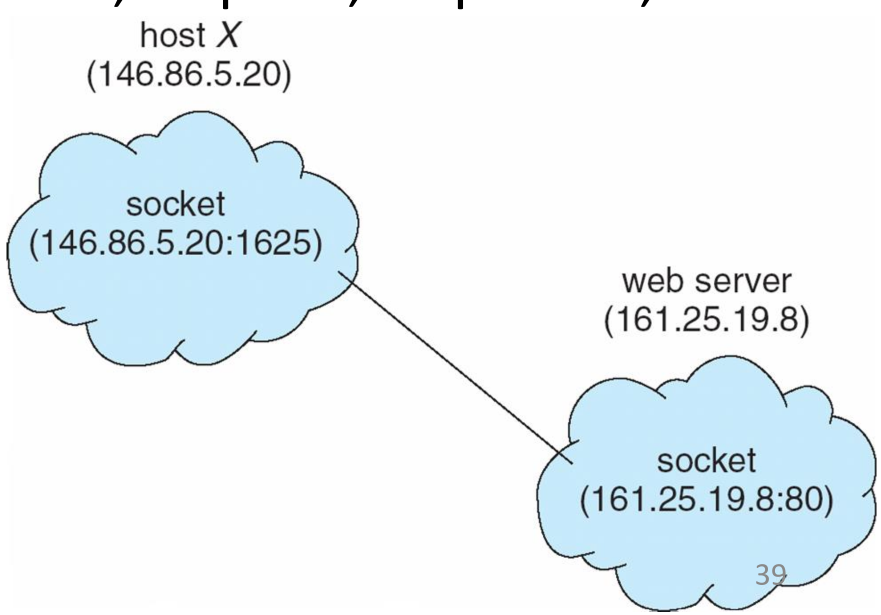
problems? How to resolve?

Link buffer
- Zero buffer (Rendezvous)
- Bounded buffer
- Unbounded buffer

Shared memory vs msg passing?

IPC - Sockets

- Communication b/w a pair of sockets
- socket: port for communication – local to each host (IP)
  - IP:port or hostname:port
  - 161.25.19.8:1625: “port 1625 on host 161.25.19.8”
- Special IP 127.0.0.1 (loopback) - local system
- Well-known ports: < 1024 - smtp: 25, http: 80, https: 443, etc
- Socket types
  - Connection-oriented (TCP)
  - Connectionless (UDP)
  - MulticastSocket – multi recipients
- Linux cmd: netstat, lsof
- Binding: C, Java, etc.



Sockets in Java (Date server/client)

javac DateClient.java
java DateClient

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args)  {
    try {
      // could be changed one other than the localhost
      Socket sock = new Socket("127.0.0.1",6013);
      InputStream in = sock.getInputStream();
      BufferedReader bin =
        new BufferedReader(new InputStreamReader(in));

      String line;
      while( (line = bin.readLine()) != null)
        System.out.println(line);

        sock.close();
    }
    catch (IOException ioe) {
        System.err.println(ioe);
    }
  }
}
```

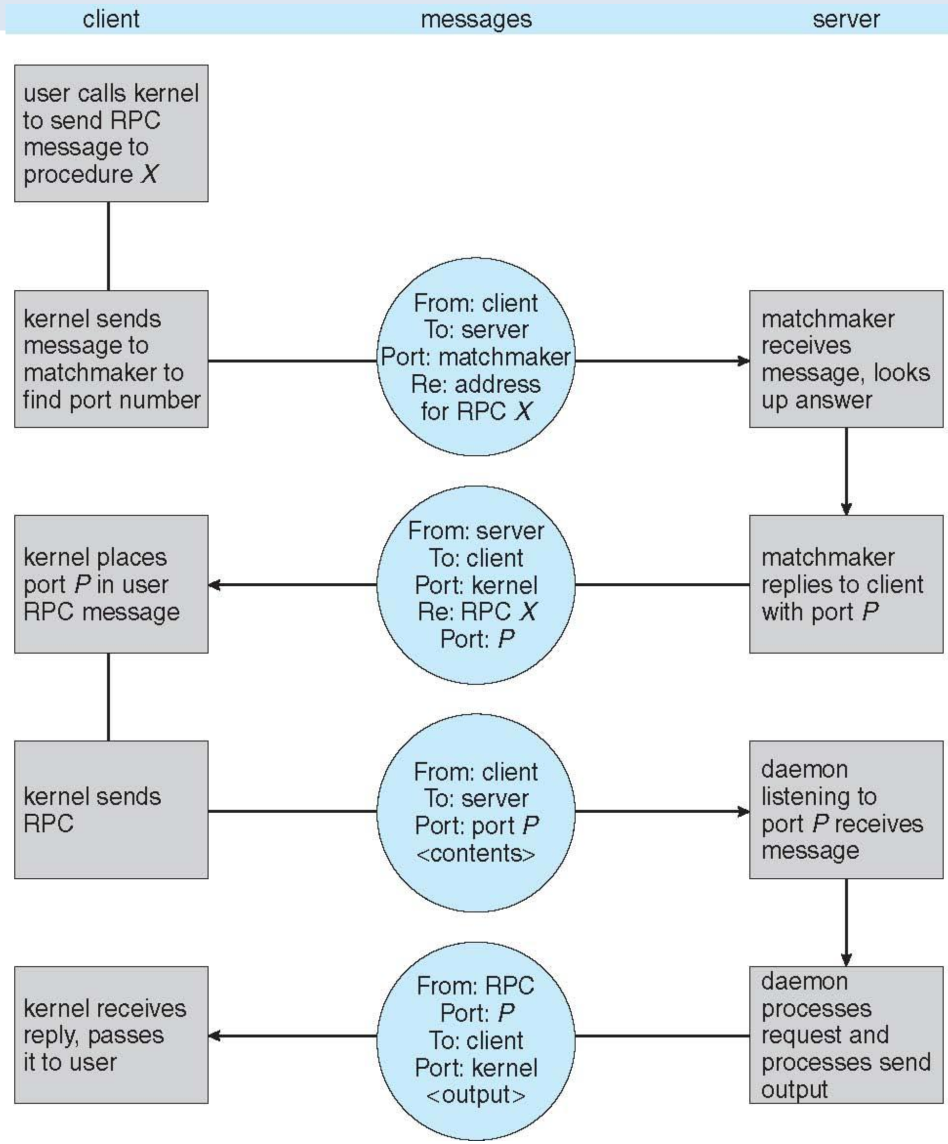
Demo!

javac DateServer.java
java DateServer

```
import java.net.*;
import java.io.*;

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      /* now listen for connections */
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        /* write the Date to the socket */
        pout.println(new java.util.Date().toString());

        /* close the socket and resume */
        /* listening for connections */
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

problem?

IPC – Remote Procedure Call (RPC)



IPC - Ordinary Pipes

- Ordinary Pipes: communication in producer-consumer style
  - Unidirectional
  - Exists only when procs are communicating
    - Producer: writes to one end (the write-end of the pipe)
    - Consumer: reads from the other end (the read-end of the pipe)
  - Require parent-child relationship b/w communicating procs
    - Parent creates pipe (special file), then forks a child that shares pipe w/ parent
    - same machine only
- Shell: `ls | more`

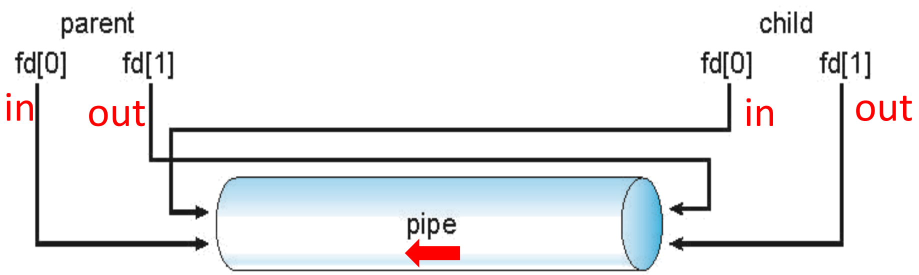
  How to program it in shell?
  - stdout of `ls` becomes stdin of `more`
- Windows calls these anonymous pipes

Example: Ordinary Pipes

Demo!

man pipe

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;
```




```c
/* create the pipe */
if (pipe(fd) == -1) {
   fprintf(stderr,"Pipe failed");
   return 1;
}

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
   fprintf(stderr, "Fork Failed");
   return 1;
}

if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s",read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}

return 0;
}
```

IPC - Named Pipes

- Named Pipes
  - Bidirectional:
    - Half-duplex: A ↔ B; not simultaneously. E.g., walkie-talkie
    - Full-duplex: A ↔ B simultaneously. E.g., telephone
  - No parent-child relationship is needed b/w communicating procs
  - >2 procs can use the named pipe for communication
  - Continue to exist even after procs have finished
- Named pipes in Linux/UNIX (aka FIFO)
  - As a file in FS; continue to exist until it is removed from FS
  - Bidirectional & half-duplex: for procs on the same machine
  - API: `mkfifo()`, `open()`, `read()`, `write()`, `close()`
- Named pipes in Windows
  - Bidirectional & full-duplex: for procs on local or remote machine
  - API: `CreateNamedPipe()`, `ConnectNamedPipe()`, `ReadFile()`, `WriteFile()`

Parallel Programming

- Get computations done in parallel
- Approaches
  - Single computer: multiple processes, multiple threads
  - Multiple computers: distributed system, cloud computing
- Issues
  - How many processes or threads?
    - Number of CPUs, nodes
  - How to partition the computation? Spit data? data dependency?
    - Granularity of a process/thread
  - How to communication among processes/threads: IPC?
  - How to coordinate among processes/threads → synchronization (later)
  - Load balance?
  - Testing and debugging?

Question

- Q: [multi-choice] Which one is true?

A. Msg passing is better suited for large volume of data exchange than shared memory

B. IPC with shared memory is limited to parent-child processes only

C. Any two independent processes can communicate with ordinary pipe

D. RPC and socket allow communication across computers and within a computer

E. None of the above

Summary

- Process state transition diagram
- PCB content?
- What does context switch do?
- What is zombie process? Orphan process? How to resolve orphan process?
- Proc life cycle: create, terminate – fork(), exec(), exit(), wait()
- User enters “cc -o a b.c” in a shell, what does argv look like when shell invokes exec()?
- IPC
  - Shared memory
  - Message passing
  - RPC
  - Sockets
  - Pipes: ordinary pipe vs named pipe

Self Exercises

- 9/E: 3.1, 3.2, 3.4, 3.5, 3.9, 3.12, 3.13, 3.14, 3.15, 3.17, 3.18, 3.21, 3.25, 3.27, Project 1 part 1 & 2
- 10/E: 3.1, 3.2, 3.4, 3.5, 3.8, 3.10, 3.11, 3.12, 3.13, 3.14, 3.16, 3.17, 3.18, 3.19, 3.21, 3.26, Project 1 part 1~5