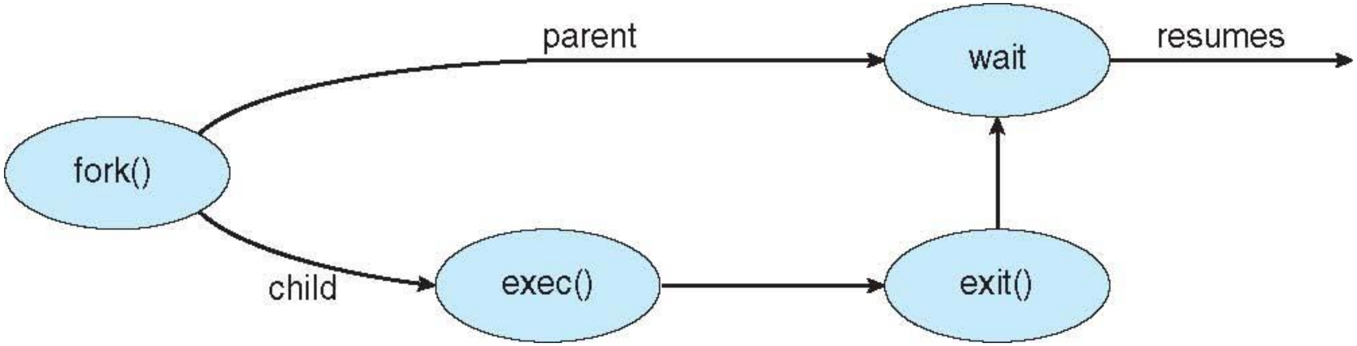
Recap

- What are the three components of a process?
  - Address space
  - CPU context
  - OS resources

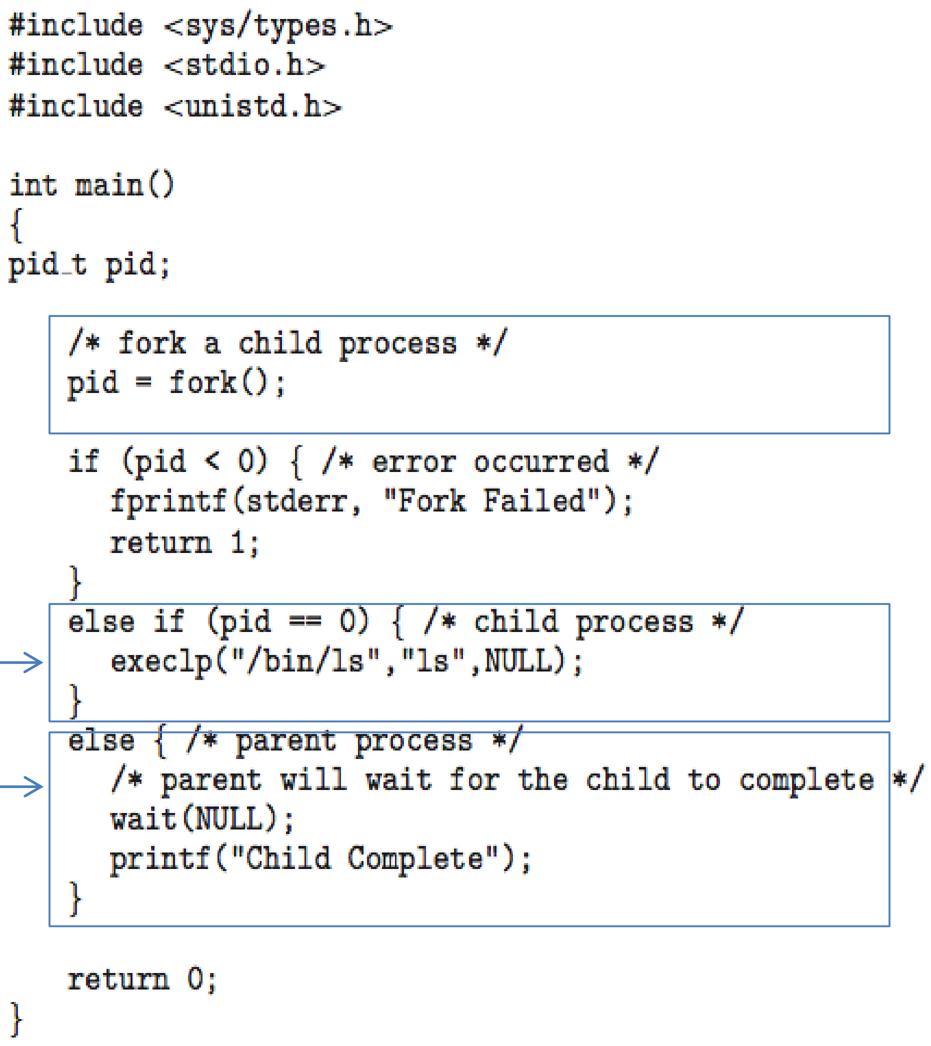
- What are the steps of a context switching?
  - Save & restore CPU context
  - Change address space and other info in the PCB

Process Creation

- UNIX examples
  - `fork()` system call creates a new process, which is **a copy of the parent process**
  - `exec()` system call used after a `fork()` to replace the process′ memory space with a new program



Example: Forking a Process in UNIX

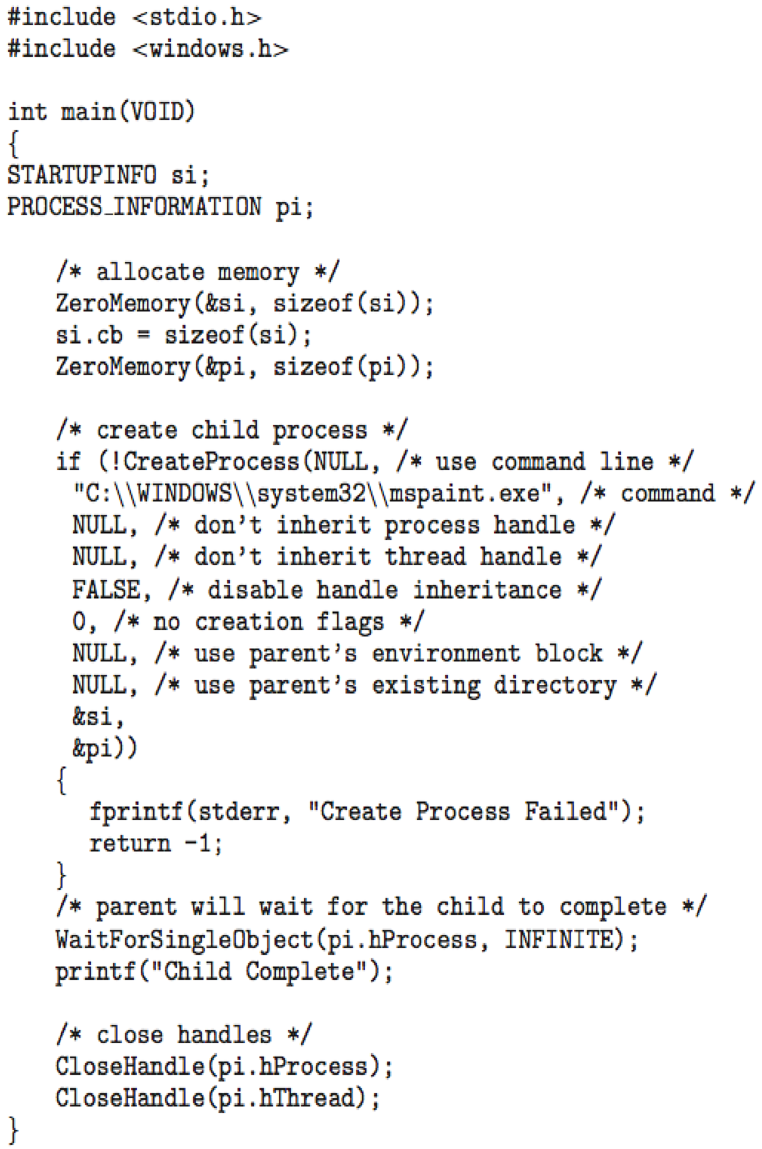


```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Child

Parent

Example: Forking a Process in Windows



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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
      "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
      NULL, /* don’t inherit process handle */
      NULL, /* don’t inherit thread handle */
      FALSE, /* disable handle inheritance */
      0, /* no creation flags */
      NULL, /* use parent’s environment block */
      NULL, /* use parent’s existing directory */
      &si,
      &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Process Termination

- Normal termination via **`exit()`** system call.
  - Exit by itself.
  - Returns status data from child to parent (via **`wait()`**)
  - Process's resources are deallocated by operating system
- Forced termination via **`kill()`** system call
  - Kill someone else (child)
- **Zombie** process
  - If no parent waiting (did not invoke **`wait()`**)
- **Orphan** process
  - If parent terminated without invoking **`wait`**
  - **Q: who will be the parent of a orphan process?**
  - **A: Init process**

Mini Quiz

```
int count = 0;
int main()
{
    int pid = fork();
    if (pid == 0){
        count++;
        printf("Child: %d\n", count);
    } else{
        wait(NULL);
        count++;
        printf("Parent: %d\n", count);
    }
    count++;
    printf("Main: %d\n", count);
    return 0;
}
```

- Hints
  - Each process has its **own private address space**
  - Wait() blocks until the child finish
- Output?

```
Child: 1
Main: 2
Parent: 1
Main: 2
```

Inter-Process Communication

Heechul Yun

Disclaimer: some slides are adopted from the book authors' slides with permission

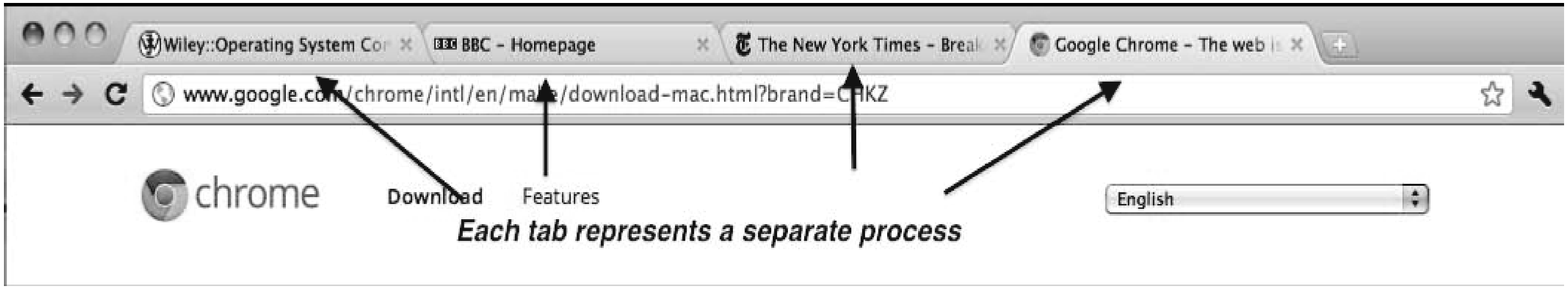
Inter-Process Communication (IPC)

- What is it?
  - Communication among processes

- Why needed?
  - Information sharing
  - Modularity
  - Speedup

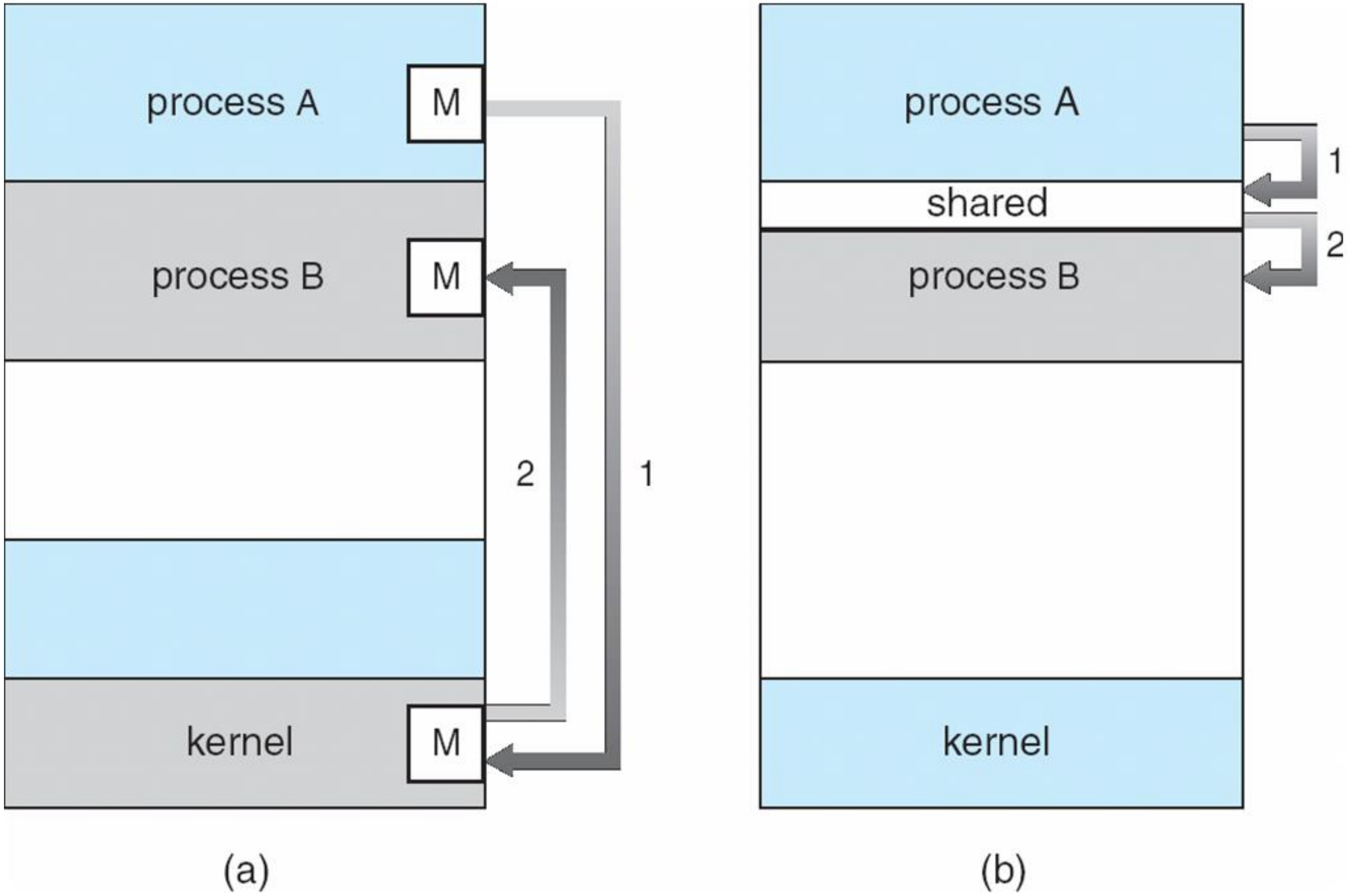
Chrome Browser

- Multi-process architecture
- Each tab is a **separate** process
  - – Why?
  - – How to communicate among the processes?


Wiley::Operating System Co
BBC - Homepage
The New York Times - Break
Google Chrome - The web i
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English

Each tab represents a separate process

Models of IPC



message passing

shared memory

Models of IPC

- Shared memory
  - share a region of memory between co-operating processes
  - read or write to the shared memory region

  ++ fast communication

  -- synchronization is very **difficult**

- Message passing
  - exchange messages (*send* and *receive*)
  - typically involves data copies (to/from buffer)

  ++ synchronization is **easier**

  -- slower communication

Interprocess Communication in Unix (Linux)

- **Pipe**
- **FIFO**
- **Shared memory**
- **Socket**
- Message queue
- …

Pipes

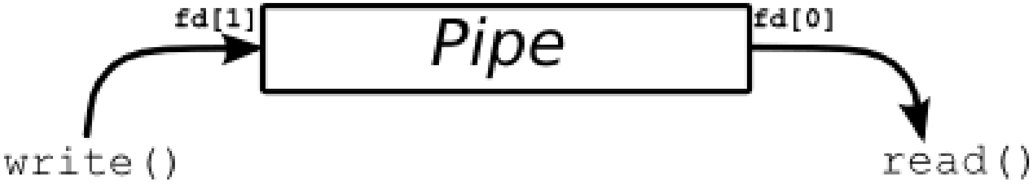
- Most basic form of IPC on all Unix systems
  - Your shell uses this a lot (and your 1st programming project too)

```
ls | more
```

- Characteristics
  - Unix pipes only allow **unidirectional** communication
  - Communication **between parent-child**
  - Processes must be in the **same OS**
  - Pipes exist only until the processes exist
  - Data can only be collected in FIFO order

IPC Example Using Pipes

```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = “Hello World\n";

  /* create a pipe */
  pipe(fds);

  /* create a new process using fork */
  if (fork() == 0) {

    /* child process. All file descriptors, including
       pipe are inherited, and copied.*/
    write(fds[1], s, strlen(s));
    exit(0);
  }

  /* parent process */
  read(fds[0], buf, strlen(s));
  write(1, buf, strlen(s));
}
```



(*) Img. source: http://beej.us/guide/bgipc/output/html/multipage/pipes.html

Pipes Used in Unix Shells

- Pipes commonly used in most Unix shells
  - output of one command is input to the next command
  - example: `ls| more`
- How does the shell realize this command?
  - create a pipe
  - create a process to run `ls`
  - create a process to run `more`
  - the standard output of the process to run `ls` is redirected to a pipe streaming to the process to run `more`
  - the standard input of the process to run `more` is redirected to be the pipe from the process running `ls`

Named Pipes (FIFO)

- Pipe with a name !
  - More powerful than anonymous pipes
  - no parent-sibling relationship required
  - FIFOs exists even after creating process is terminated
- Characteristics of FIFOs
  - appear as typical *files*
  - communicating process must reside on the same machine

Example: Producer

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // create FIFO file
  fd = open(FIFO_NAME, O_WRONLY); // open FIFO for writing

  printf("Enter text to write in the FIFO file: ");
  fgets(str, MAX_LENGTH, stdin);
  while(!(feof(stdin))){
    if ((num = write(fd, str, strlen(str))) == -1)
      perror("write");
    else
      printf("producer: wrote %d bytes\n", num);
    fgets(str, MAX_LENGTH, stdin);
  }
}
```

Example: Consumer

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // make fifo, if not already present
  fd = open(FIFO_NAME, O_RDONLY); // open fifo for reading

  do{
    if((num = read(fd, str, MAX_LENGTH)) == -1)
      perror("read");
    else{
      str[num] = '\0';
      printf("consumer: read %d bytes\n", num);
      printf("%s", str);
    }
  }while(num > 0);
}
```