Linux System Programming

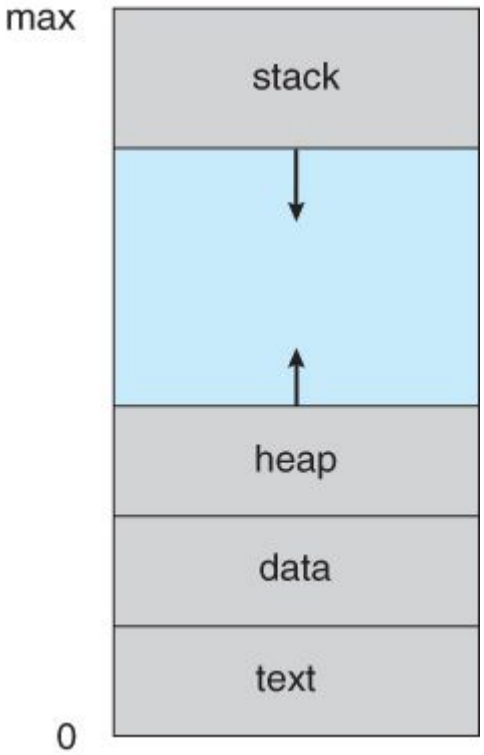
by ProgCoach4U

프로세스와 쓰레드

Program/Process/Thread

- Program
  - 실행 가능한 코드, 바이너리, 파일로 저장
- Process
  - 실행 중인 프로그램
  - 프로그램 이미지, 메모리 인스턴스, 커널 리소스 등의 정보
  - 하나 이상의 쓰레드
- Thread
  - 프로세스 내의 실행 단위
  - 가상화 된 프로세서, 스택, 레지스터, 명령어 포인터 등 프로세서의 상태 포함
  - 프로세스 내의 모든 쓰레드는 같은 주소 공간을 공유

Process in memory



함수 지역 변수, 함수 호출/리턴

동적 메모리 영역

전역 변수 등

compile 된 프로그램 코드

Single-thread vs. Multi-thread

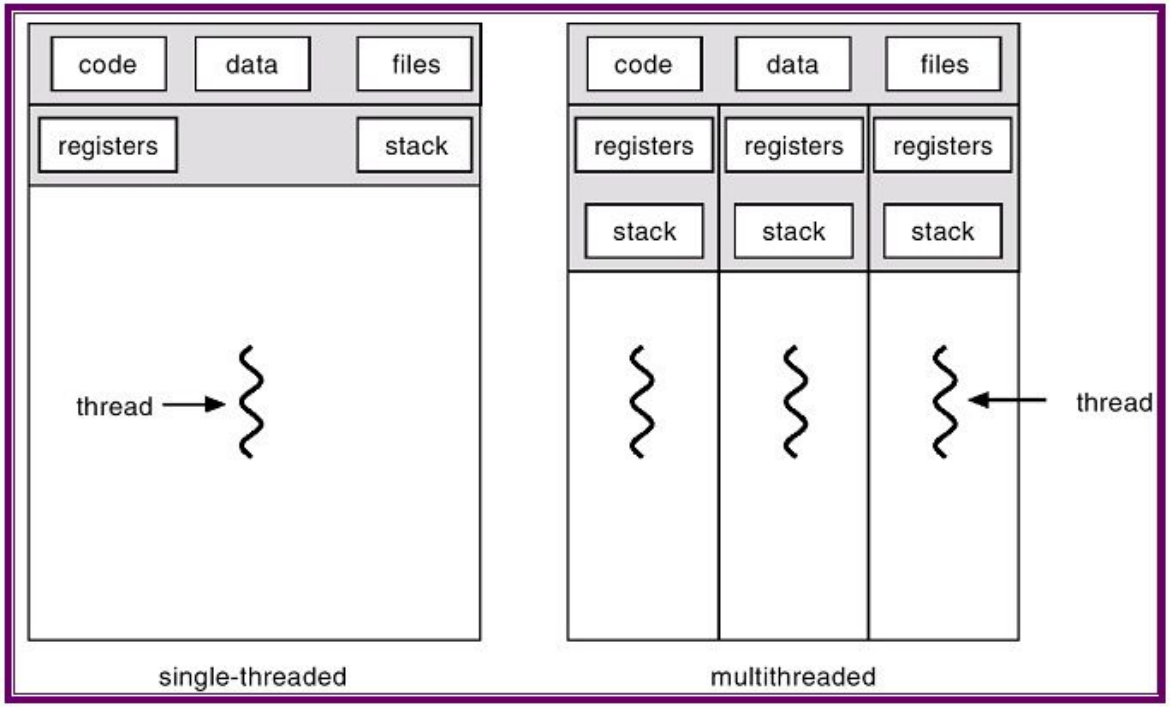


image source: http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

Process Hierarchy

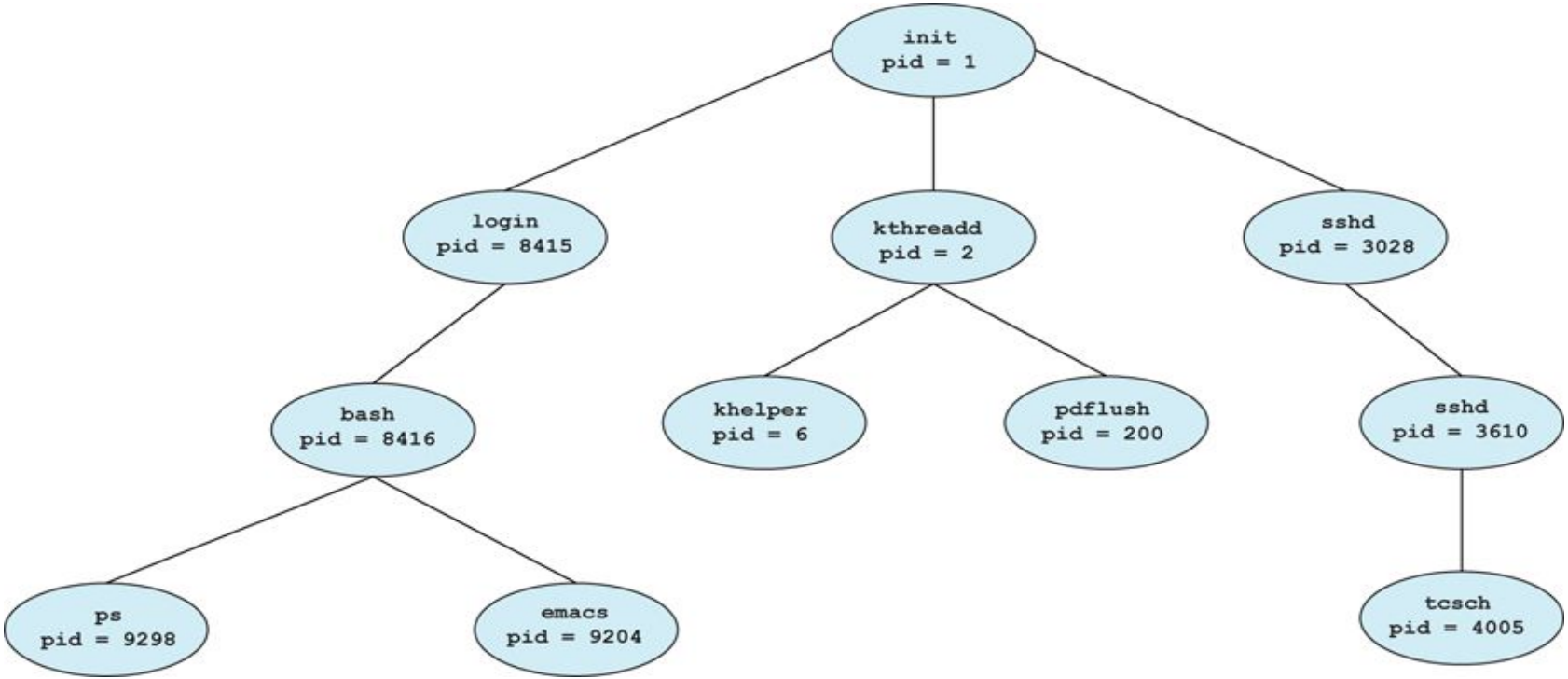


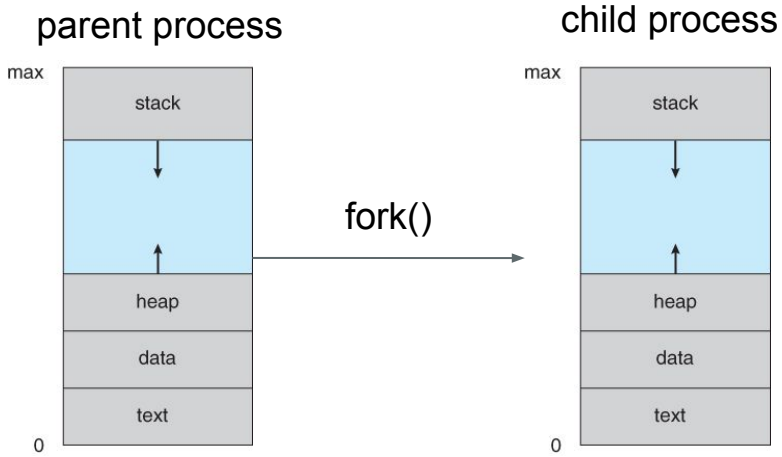
image source: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

프로세스 복제/생성 - fork()

```
pid_t fork(void);
```

파라미터
- 없음

반환값
- 성공시
  - parent process: Child process의 PID
  - child process: 0
- 실패시
  - parent process: -1
  - child process: 생성되지 않음



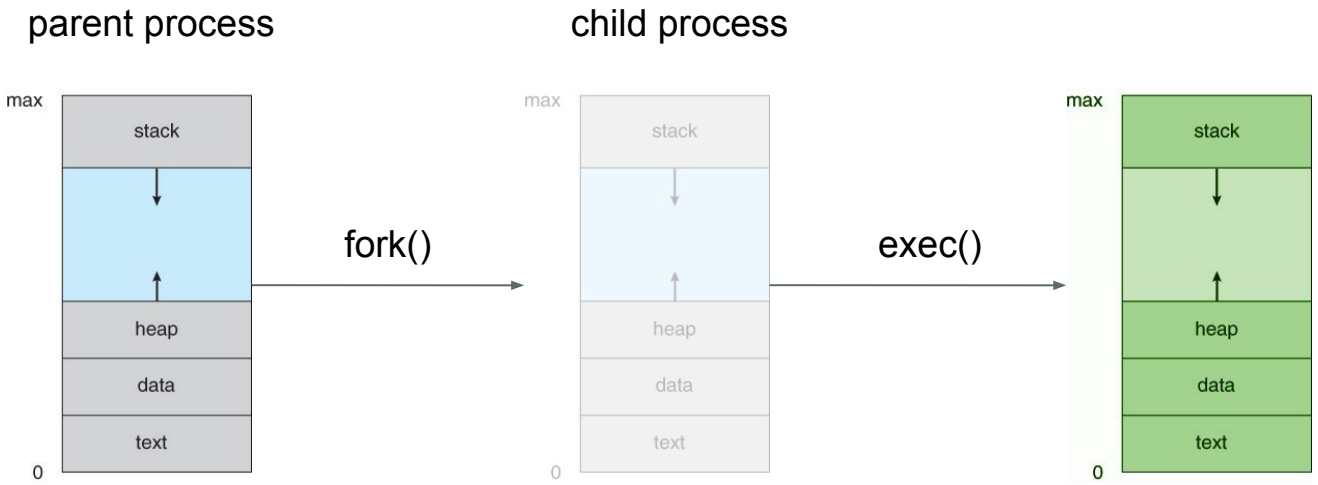
pid/ppid 얻기 - getpid(), getppid()

```
pid_t getpid(void);
pid_t getppid(void);
```

파라미터
- 없음

반환값
- getpid(): 현재 프로세스의 pid
- getppid() 현재 프로세스의 부모 프로세스 pid

새로운 프로그램 실행 - exec()



exec APIs

```
int execl(const char *path, const char *arg, ... /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *arg, ... /*, (char *) NULL, char *
const envp[] */);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

execl: list based arguments
execv: vector(char pointer array) based arguments
suffix 'p': $PATH에서 경로 찾음
suffix 'e': 새로운 환경 변수 설정
반환값: 실패시에만 -1 리턴

프로세스 종료 - exit()

```
void exit(int status);
```

파라미터
- status: exit status
  - 0: 성공
  - non-zero: 실패

좀비 프로세스

좀비 프로세스란?

- 종료 처리 중 멈춰있는 상태의 프로세스
- 자식 프로세스가 **종료되었지만** 부모 프로세스가 해당 프로세스의 종료에 대해 처리하지 않아 **커널 프로세스 테이블에 남아있는 상태**의 프로세스
- 좀비 프로세스가 생성되면 커널 프로세스 테이블을 점유 → **시스템 리소스 장악**
- 좀비 프로세스를 만들지 않으려면? **부모 프로세스가 자식 프로세스에 대해 종료 처리를 해야 한다.**

자식 프로세스 종료 시그널 - SIGCHLD

- 언제? 자식 프로세스가 종료되었을 때
- 누구에게? 부모 프로세스에게
- 무엇을? SIGCHLD 시그널
- 어떻게? signal 전송

부모 프로세스는 SIGCHLD에 대해 수신 대기

SIGCHLD를 수신하면 자식 프로세스 상태 확인 → 종료된 자식 프로세스 처리

자식 프로세스 종료 대기 - wait()

```
pid_t wait(int *wstatus);
```

파라미터
- wstatus: child process의 종료 상태

반환값
- 성공시 - terminated 된 자식 프로세스의 pid
- 실패시 - -1

```
WIFEXITED(wstatus)
WEXITSTATUS(wstatus)

WIFSIGNALED(wstatus)
WTERMSIG(wstatus)

WCOREDUMP(wstatus)

WIFSTOPPED(wstatus)
WSTOPSIG(wstatus)
WIFCONTINUED(wstatus)
```

자식 프로세스 종료 대기 - waitpid()

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

파라미터
- pid: 종료 대기할 프로세스의 pid
- status: 자식 프로세스의 종료 코드
- options
  - WNOHANG
  - WUNTRACED
  - WCONTINUED

반환값
- 양수: 상태가 바뀐 child process의 pid
- 0: WNOHANG 지정시
- -1: 실패

자식 프로세스 종료 대기 - wait4()

```
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

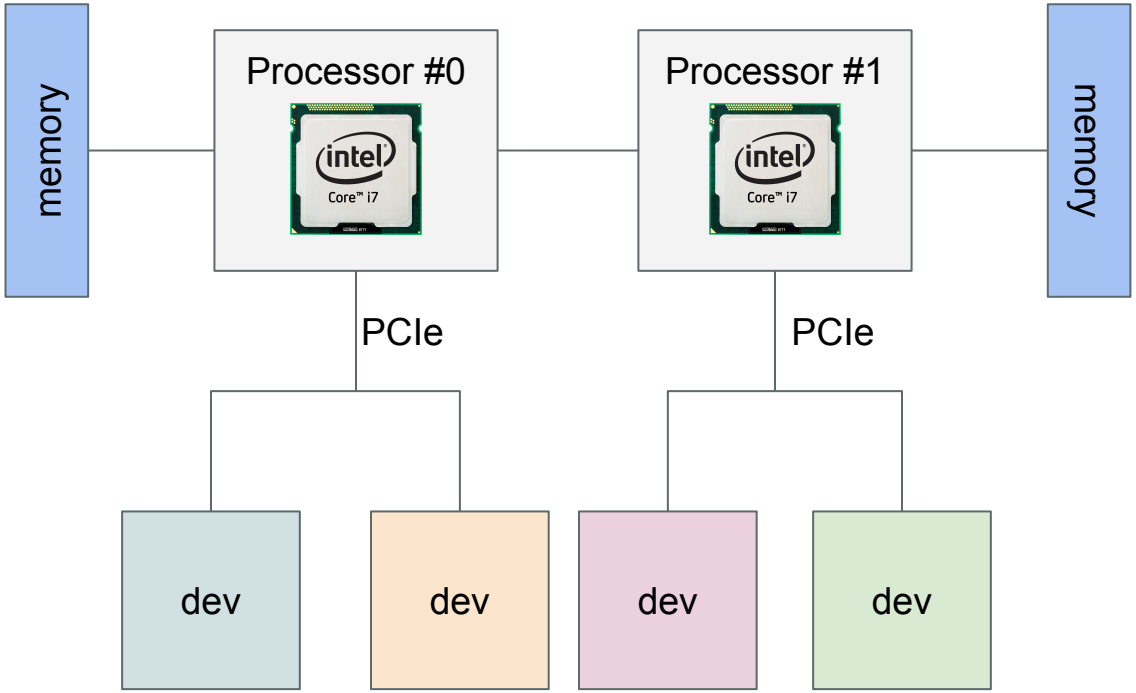
파라미터
- pid: 종료 대기할 프로세스의 pid
- status: 자식 프로세스의 종료 코드
- options
  - WNOHANG
  - WUNTRACED
  - WCONTINUED
- rusage: 리소스 사용량

반환값
- 양수: 상태가 바뀐 child process의 pid
- 0: WNOHANG 지정시
- -1: 실패
-

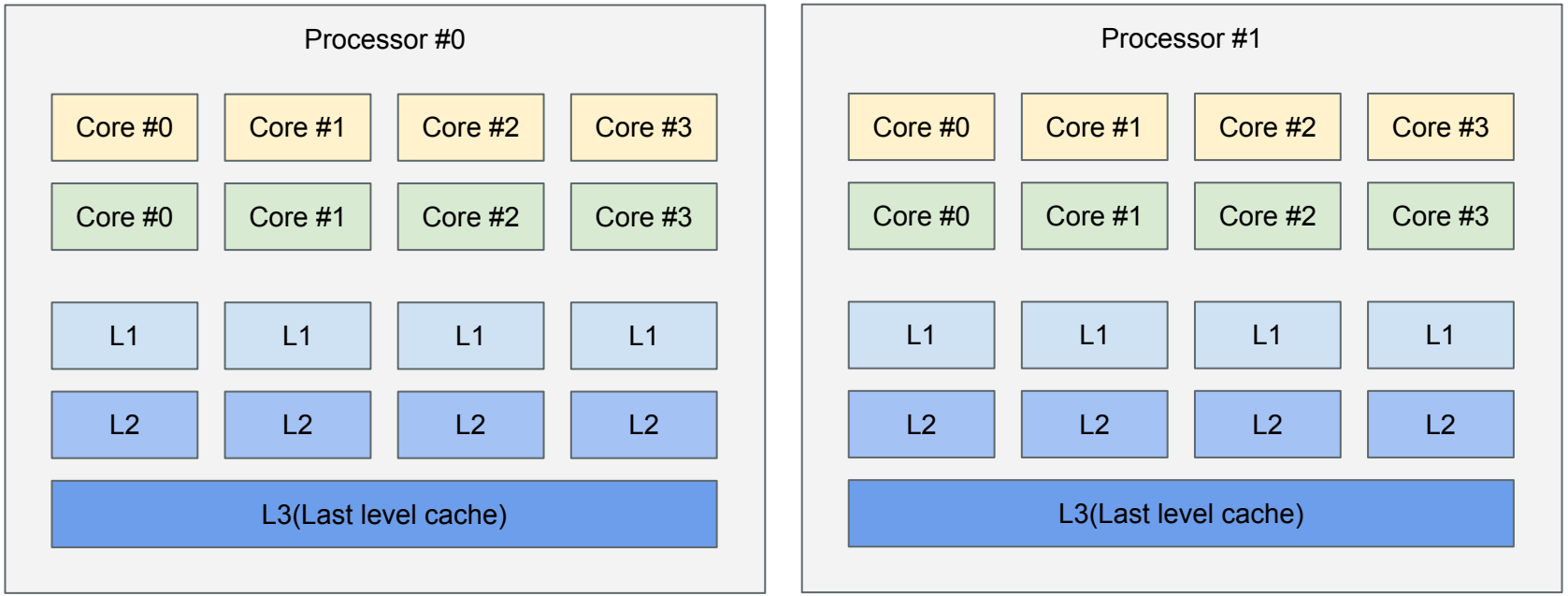
자식 프로세스 종료 대기 - wait4()

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long   ru_maxrss;        /* maximum resident set size */
    long   ru_ixrss;         /* integral shared memory size */
    long   ru_idrss;         /* integral unshared data size */
    long   ru_isrss;         /* integral unshared stack size */
    long   ru_minflt;        /* page reclaims (soft page faults) */
    long   ru_majflt;        /* page faults (hard page faults) */
    long   ru_nswap;         /* swaps */
    long   ru_inblock;       /* block input operations */
    long   ru_oublock;       /* block output operations */
    long   ru_msgsnd;        /* IPC messages sent */
    long   ru_msgrcv;        /* IPC messages received */
    long   ru_nsignals;      /* signals received */
    long   ru_nvcsw;         /* voluntary context switches */
    long   ru_nivcsw;        /* involuntary context switches */
};
```

Multi-Processor/Multi-core/Multi-thread



Multi-Processor/Multi-core/Multi-thread



프로세서 친화(Processor affinity)

```
int sched_setaffinity(pid_t pid, size_t cpusetsize, const cpu_set_t *mask);
int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
```

파라미터
- pid: 대상 프로세스의 pid
- cpusetsize: mask의 사이즈(sizeof(cpu_st_t))
- mask: CPU mask

반환값
- 성공시 0
- 실패시 -1

```
int sched_getcpu(void);
```

반환값
- 성공시 CPU 번호(non-negative)
- 실패시 -1

Single-thread vs. Multi-thread

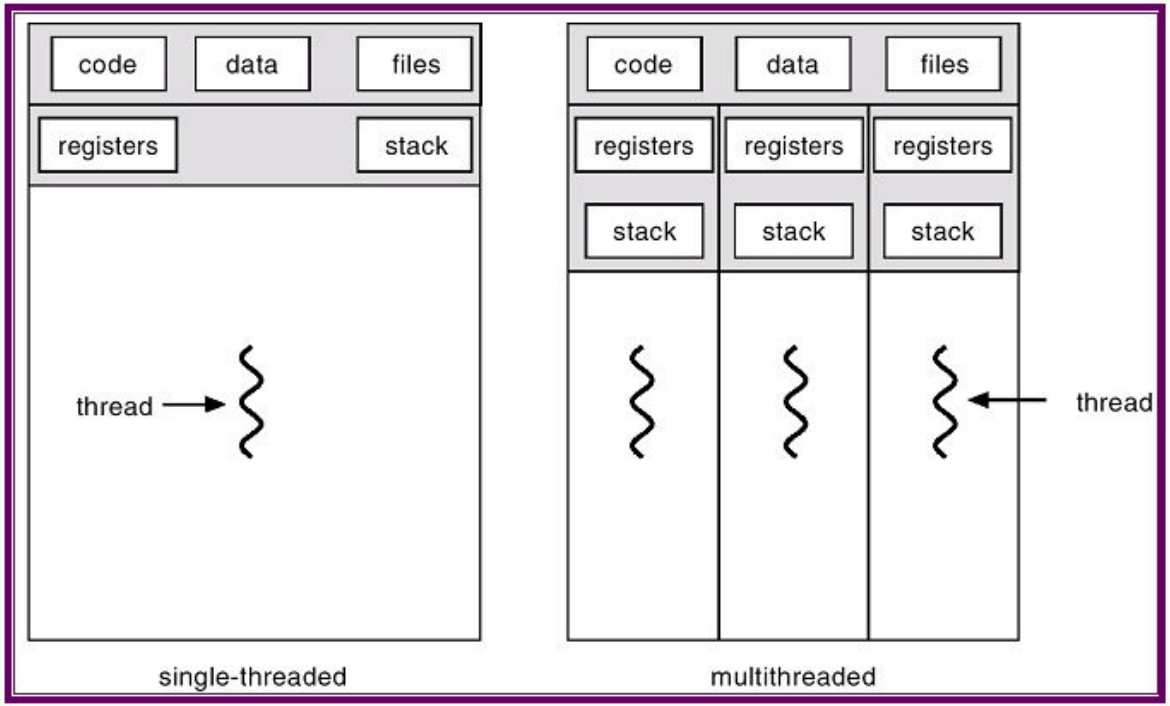


image source: http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

쓰레드 생성/종료

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

파라미터
- thread: 생성된 thread ID
- attr: 쓰레드 속성(pthread_attr_init()으로 초기화)
- start_routine: thread main function
- arg: thread main function 호출 시 사용할 파라미터

반환값
- 성공시 0
- 실패시 errno를 리턴

```
void pthread_exit(void *retval);
```

파라미터
- retval: exit status를 저장

쓰레드 조인

```
int pthread_join(pthread_t thread, void **retval);
```

- 해당 쓰레드를 종료 처리

파라미터

- thread: 기다릴 thread ID
- retval: 해당 thread의 exit status를 저장

반환값

- 성공시 0
- 실패시 errno를 리턴

쓰레드 떼어내기

int pthread_detach(pthread_t thread);
- 해당 쓰레드는 종료시 자동으로 리소스 해제됨(join이 필요 없음)

파라미터
- thread: 떼어낼 thread ID

반환값
- 성공시 0
- 실패시 errno를 리턴

쓰레드 동기화

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
           const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

파라미터
- mutex: mutex instance
- attr: mutex 속성

반환값
- 성공시 0
- 실패시 errno를 리턴

쓰레드 동기화

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

파라미터
- mutex: mutex instance

반환값
- 성공시 0
- 실패시 errno를 리턴

쓰레드 동기화

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

파라미터
- mutex: mutex instance

반환값
- 성공시 0
- 실패시 errno를 리턴

감사합니다.