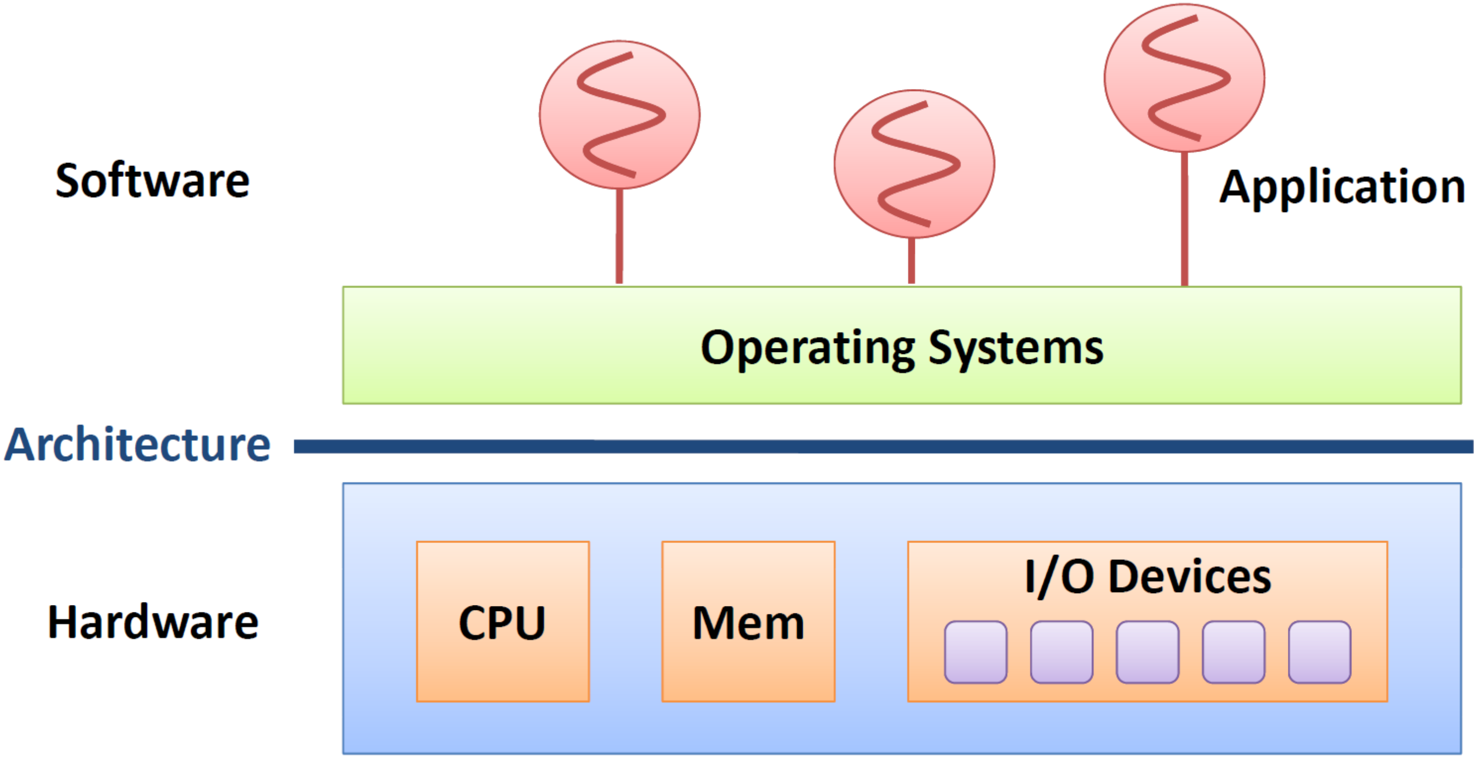
OPERATING SYSTEM REVIEW (PROCESS)

Jo, Heeseung

Operating system?

Computer systems internals



PROCESSES

Jo, Heeseung

What Is The Process?

Program?

vs.

Process?

vs.

Processor?

vs.

Task? Job?

Process Concept (1)

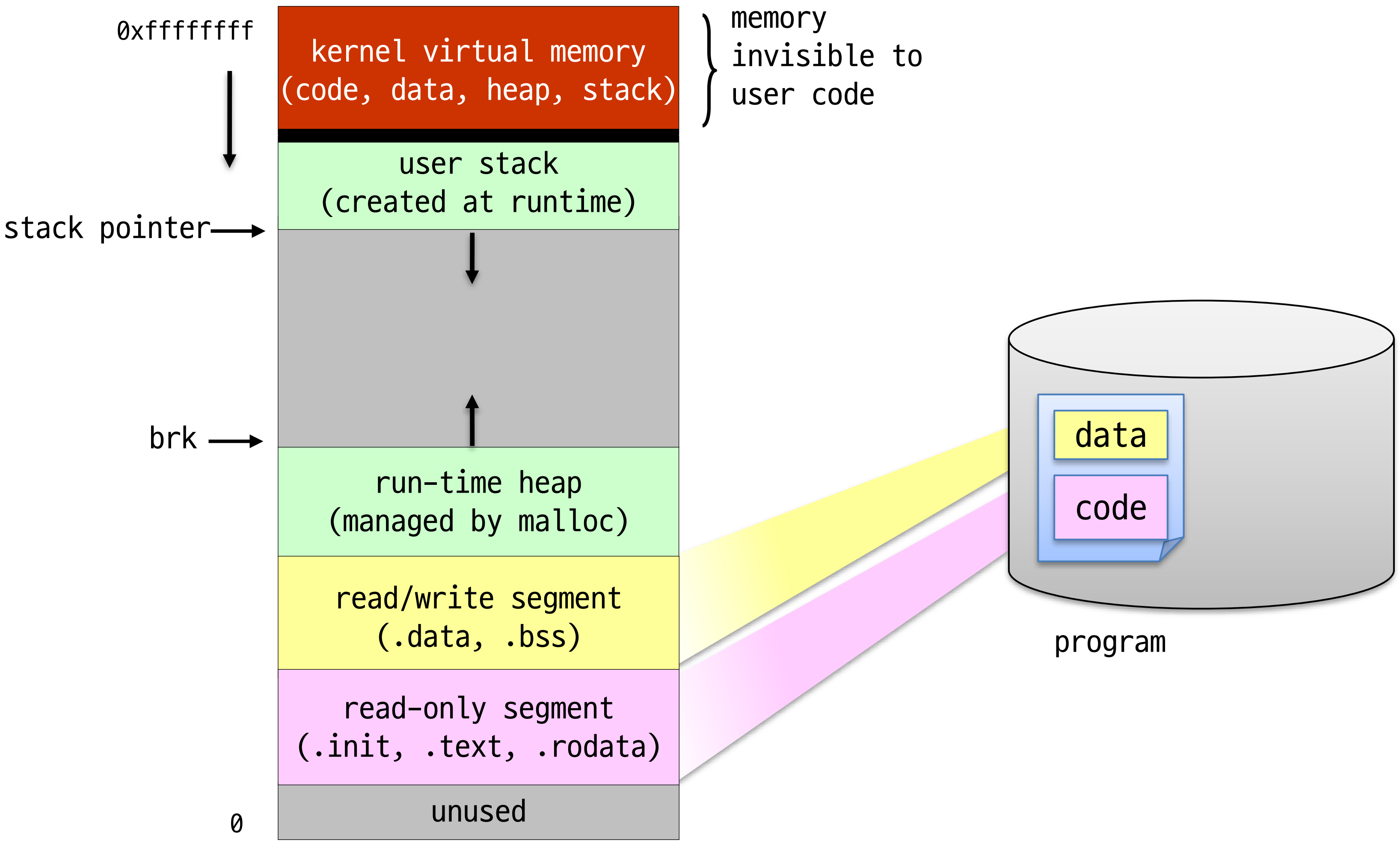
What is the process?

- An instance of a program in execution
- An encapsulation of the flow of control in a program
- A dynamic and active entity
- The basic unit of execution and scheduling
- A process is named using its process ID (PID)

- A process includes:
  - CPU contexts (registers)
  - OS resources (memory, open files, etc.)
  - Other information (PID, state, owner, etc.)

Process Concept (2)

Process in memory

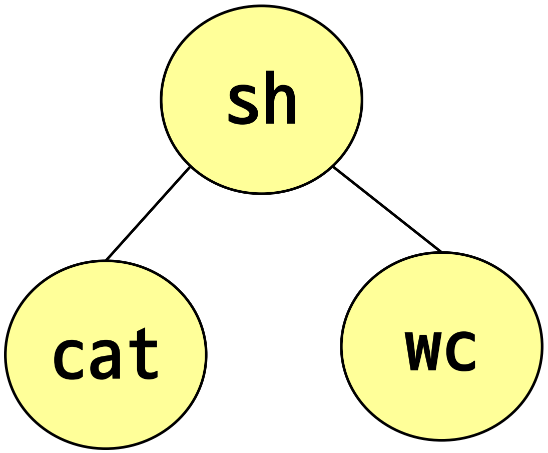


Process Creation (1)

Process hierarchy

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

$ cat file1 ¦ wc



Process Creation (2)

Process creation events

- Calling a system call
  - fork() in POSIX, CreateProcess() in Win32
  - Shells or GUIs use this system call internally
- System initialization
  - *init* process
  - PID 1 process

Process Creation (3)

Resource sharing

- Parent may inherit all or a part of resources and privileges for its children
  - UNIX: User ID, open files, etc.

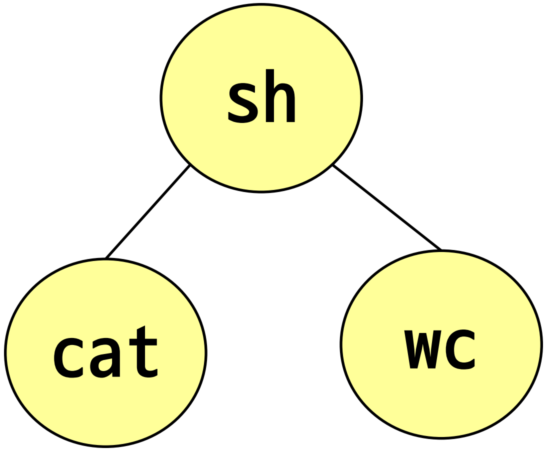
Execution

- Parent may either wait for it to finish, or it may continue in parallel

Address space

- Child duplicates the parent's address space or has a program loaded into it

```
$ cat file1 ¦ wc
```



Process Termination

Process termination events

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
  - Exceed allocated resources
  - Segmentation fault
  - Protection fault, etc.
- Killed by another process (involuntary)
  - By receiving a signal

```
#include <stdio.h>

int main()
{
    int i, fd;
    char buf[100];

    fd=open("a.txt", "r");
    if (fd==NULL)
        return -1;
    read(fd, buf, 1000);

    return 0;
}
```

fork()

fork() system call

- Creating a child process
- Copy the whole virtual address space of parent to create a child process
- Copy internal data structures to manage a child process

- Parent get the pid of a child
- Child get 0 value

fork()

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

int main()
{
	int pid;

	pid = fork();
	if (pid == 0)
	   /* child */
	   printf ("Child of %d is %d\n",
	             getppid(), getpid());
	else
	   /* parent */
	   printf ("I am %d. My child is %d\n",
	             getpid(), pid);
}
```

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

int main()
{
	int pid;

	pid = fork();
	if (pid == 0)
	   /* child */
	   printf ("Child of %d is %d\n",
	             getppid(), getpid());
	else
	   /* parent */
	   printf ("I am %d. My child is %d\n",
	             getpid(), pid);
}
```

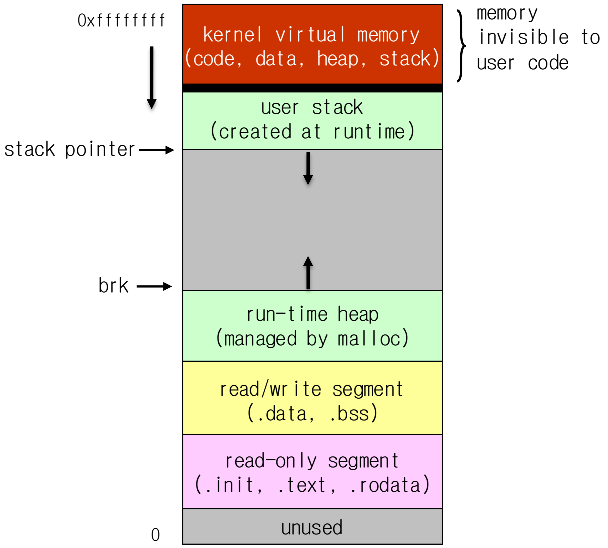
fork(): Example Output

```
% ./a.out
I am 30000. My child is 30001.
Child of 30000 is 30001.

% ./a.out
Child of 30002 is 30003.
I am 30002. My child is 30003.
```

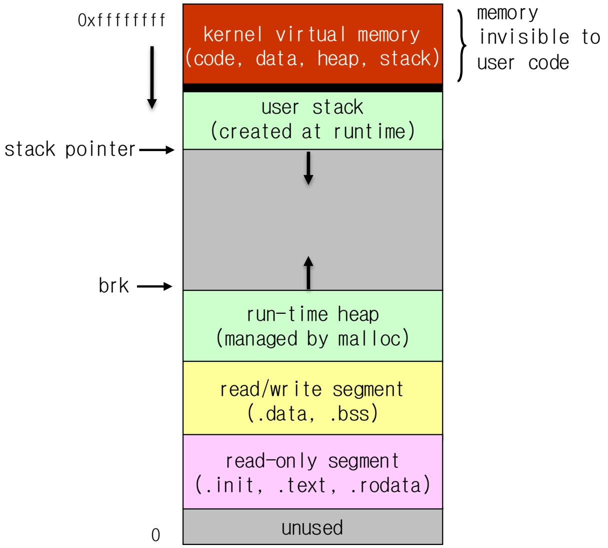
```
#include <sys/types.h>
#include <unistd.h>

int main()
{
     int pid;

     pid = fork();
     if (pid == 0)
        /* child */
        printf ("Child of %d is %d\n",
                  getppid(), getpid());
     else
        /* parent */
        printf ("I am %d. My child is %d\n",
                  getpid(), pid);
}
```

fork() and Virtual Address Space

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

int main()
{
     int pid;

     pid = fork();
     if (pid == 0)
        /* child */
        printf ("Child of %d is %d\n",
                 getppid(), getpid());
     else
        /* parent */
        printf ("I am %d. My child is %d\n",
                 getpid(), pid);
}
```



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

int main()
{
     int pid;

     pid = fork();
     if (pid == 0)
        /* child */
        printf ("Child of %d is %d\n",
                 getppid(), getpid());
     else
        /* parent */
        printf ("I am %d. My child is %d\n",
                 getpid(), pid);
}
```



Why fork()?

Very useful when the child ...

- Is cooperating with the parent
- Relies upon the parent's data to accomplish its task
- Example: Web server

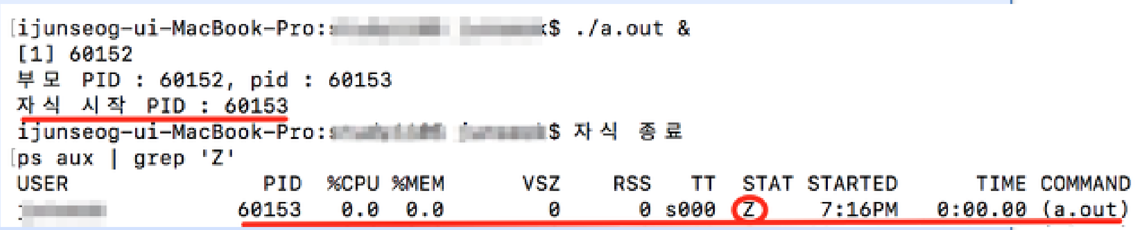
```
While (1) {
     int sock = accept();
     if ((pid = fork()) == 0) {
          /* Handle client request */
     } else {
          /* Close socket */
     }
}
```

Zombie vs. orphan process

Zombie process (defunct process)

- A process that completed execution (via the exit system call) but still has an entry in the process table
- This occurs for the child processes, where the entry is still needed to allow the parent process to read its child's exit status

```
int main() {

    pid_t childPid;

    childPid = fork();

    if (childPid > 0) {  // parent process
        printf("parent PID : %ld, pid : %d\n",(long)getpid(), childPid);
        sleep(30);
        printf("parent exit\n");
        exit(0);
    }
    else if (childPid == 0){  // 자식 코드
        printf("child PID : %ld\n", (long)getpid());
        sleep(1);
        printf("child exit\n");
        exit(0);
    }
    return 0;
}
```

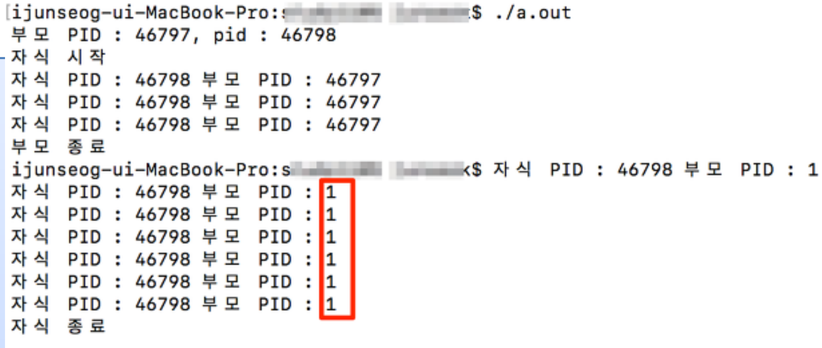


```
ijunseog-ui-MacBook-Pro:[redacted]$ ./a.out &
[1] 60152
부모 PID : 60152, pid : 60153
자식 시작 PID : 60153
ijunseog-ui-MacBook-Pro:[redacted]$ 자식 종료
ps aux | grep 'Z'
USER               PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
[redacted]       60153   0.0  0.0        0      0 s000  Z      7:16PM   0:00.00 (a.out)
```

Zombie vs. orphan process

Orphan process

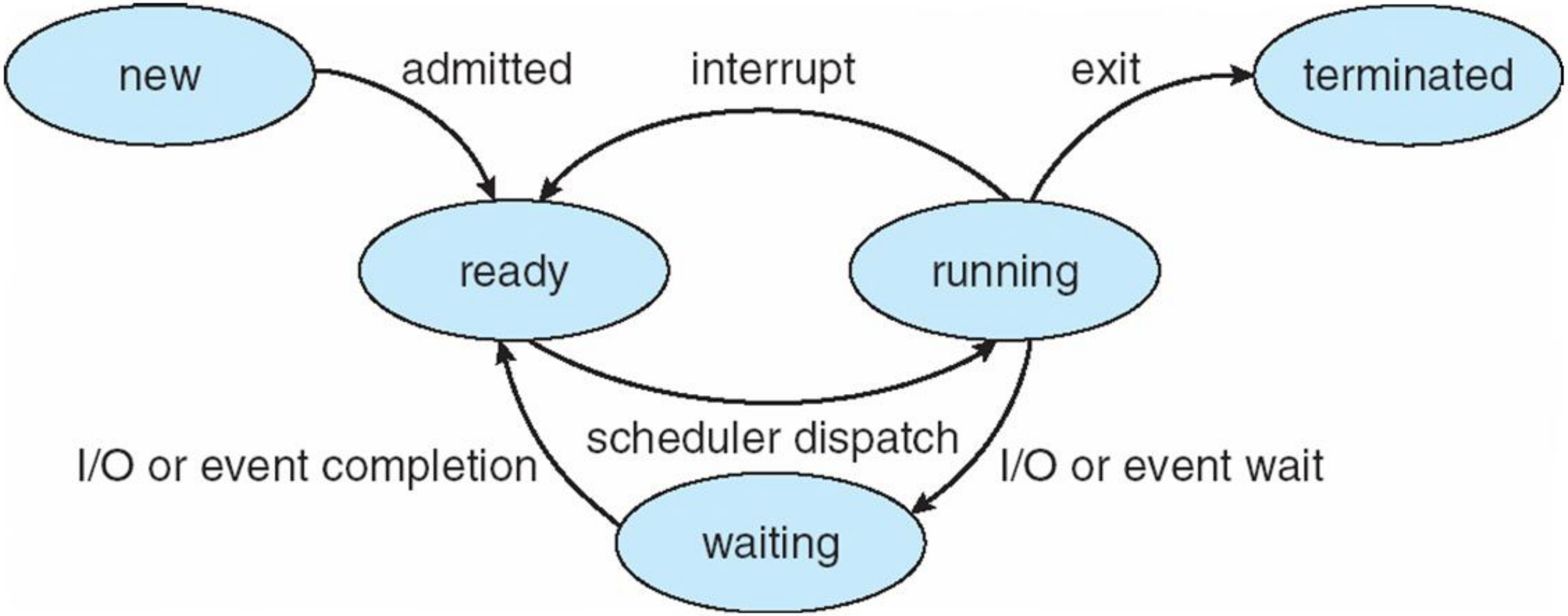
- A process whose parent process has finished or terminated, though it remains running itself
- Any orphaned process will be immediately adopted by the special init system process



```
ijunseog-ui-MacBook-Pro:[redacted]$ ./a.out
부모 PID : 46797, pid : 46798
자식 시작
자식 PID : 46798 부모 PID : 46797
자식 PID : 46798 부모 PID : 46797
자식 PID : 46798 부모 PID : 46797
부모 종료
ijunseog-ui-MacBook-Pro:[redacted]$ 자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 PID : 46798 부모 PID : 1
자식 종료
```

```c
int main() {

    pid_t childPid;
    int i;

    childPid = fork();

    if (childPid > 0) {  // parent process
        printf("parent PID : %ld, pid : %d\n",(long)getpid(), childPid);
        sleep(2);
        printf("parent exit\n");
        exit(0);
    }
    else if (childPid == 0){  // child process
        for(i=0;i<10;i++) {
            printf("child PID : %ld parent PID : %ld\n",(long)getpid(), (long)getppid());
            sleep(1);
        }
        printf("child exit\n");
        exit(0);
    }
```

Process State Transition (1)



THREADS

Jo, Heeseung

Rethinking Processes

What's similar in these cooperating processes?

- They all use (share?) the same code and data (address space)
- They all use the same privilege
- They all use the same resources (files, sockets, etc.)

What's different?

- Each has its own hardware execution state:
  PC, registers, SP, and stack

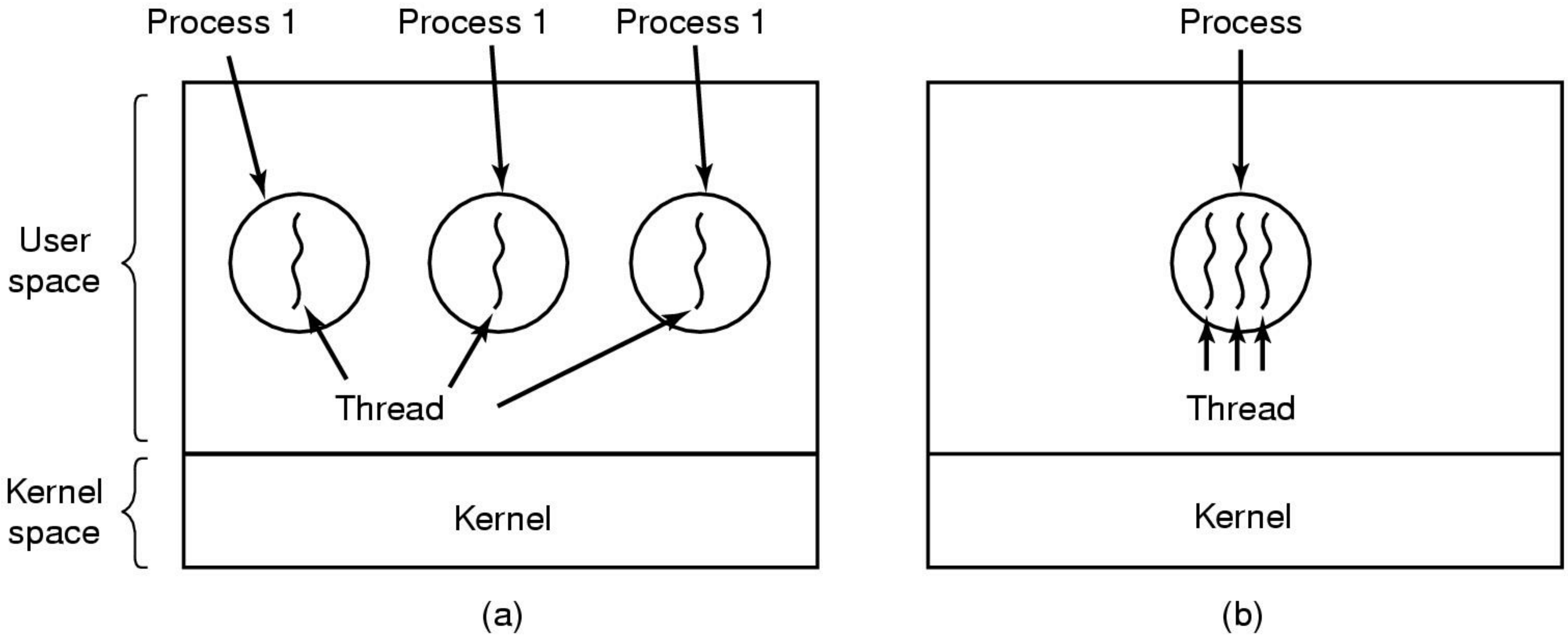
Key Idea (1)

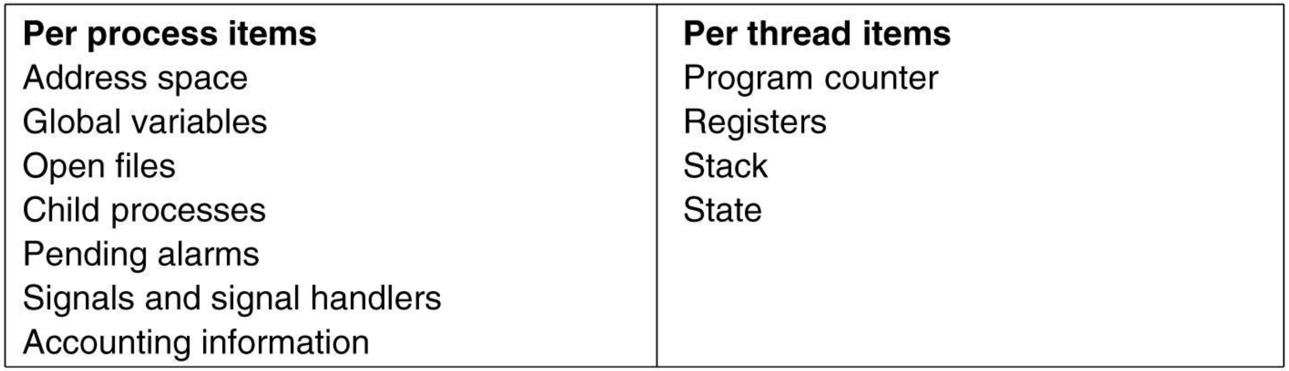
Separate the concept of a process from its execution state

- Process: address space, resources, other general process attributes
  - e.g., privileges
- Execution state: PC, SP, registers, etc.

- This execution state is usually called
  - Thread
  - Lightweight process (LWP)
  - Thread of control

Key Idea (2)

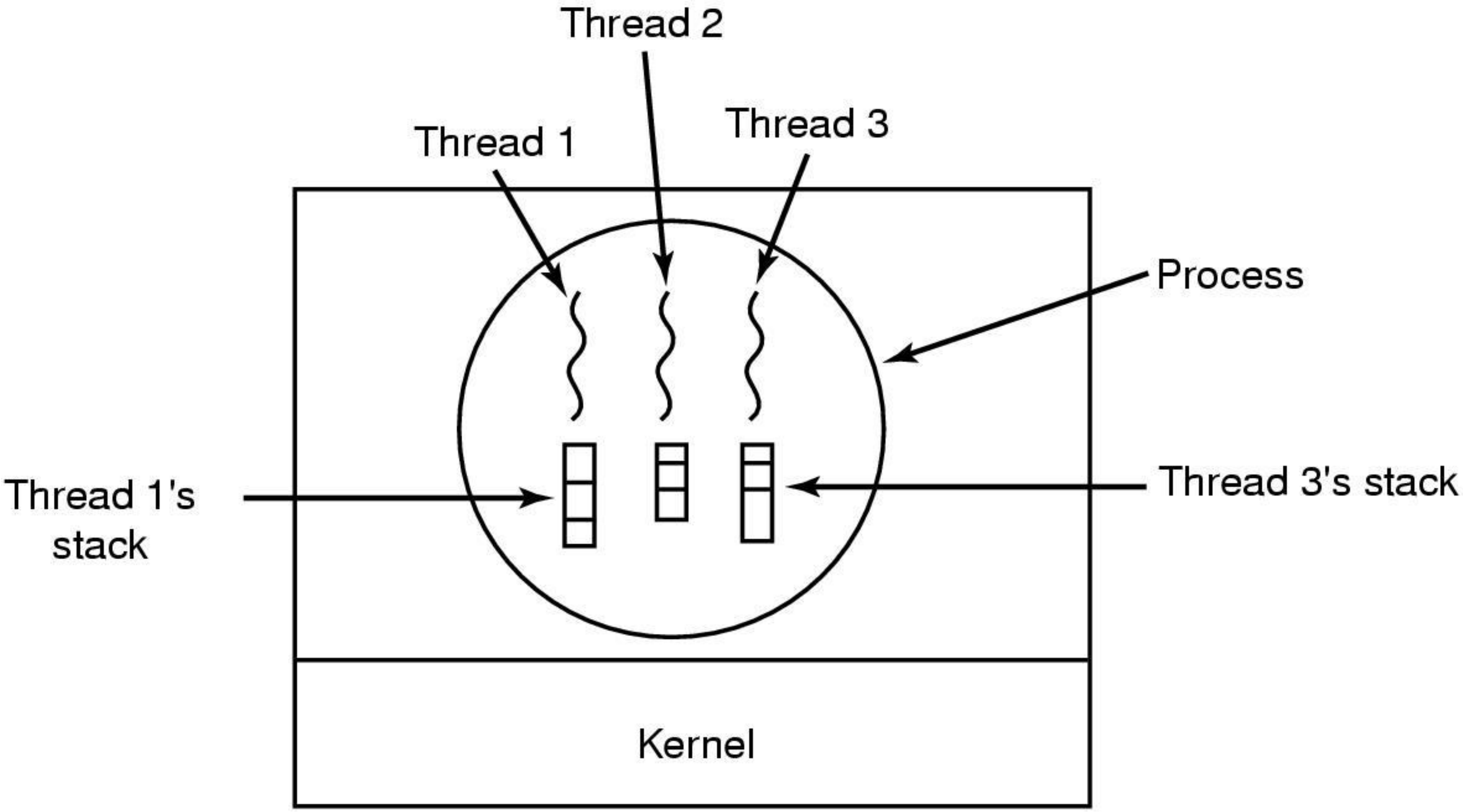




Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

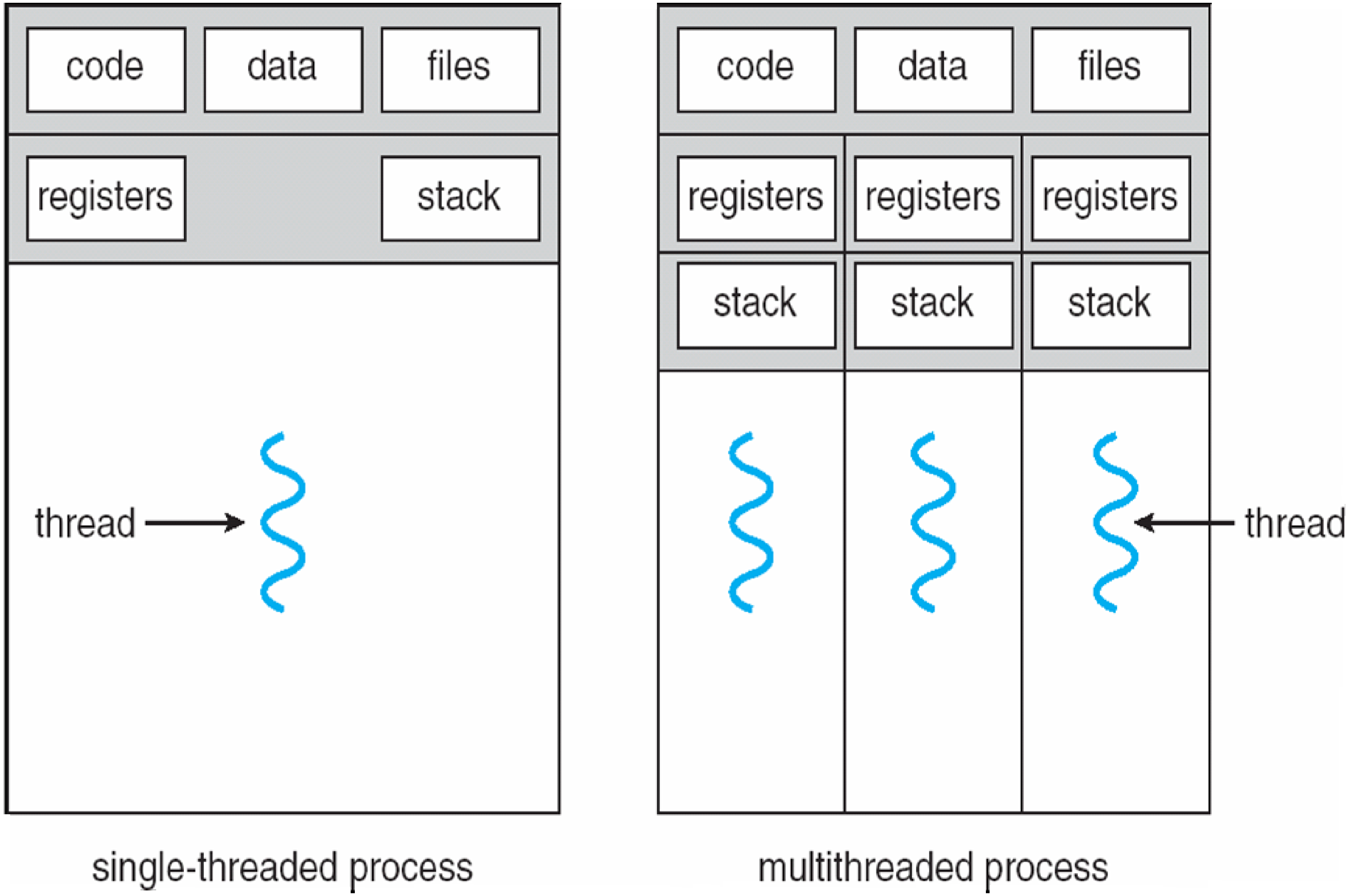
Key Idea (3)

Each thread has its own stack



Key Idea (4)

Each thread has its own stack



What is a Thread?

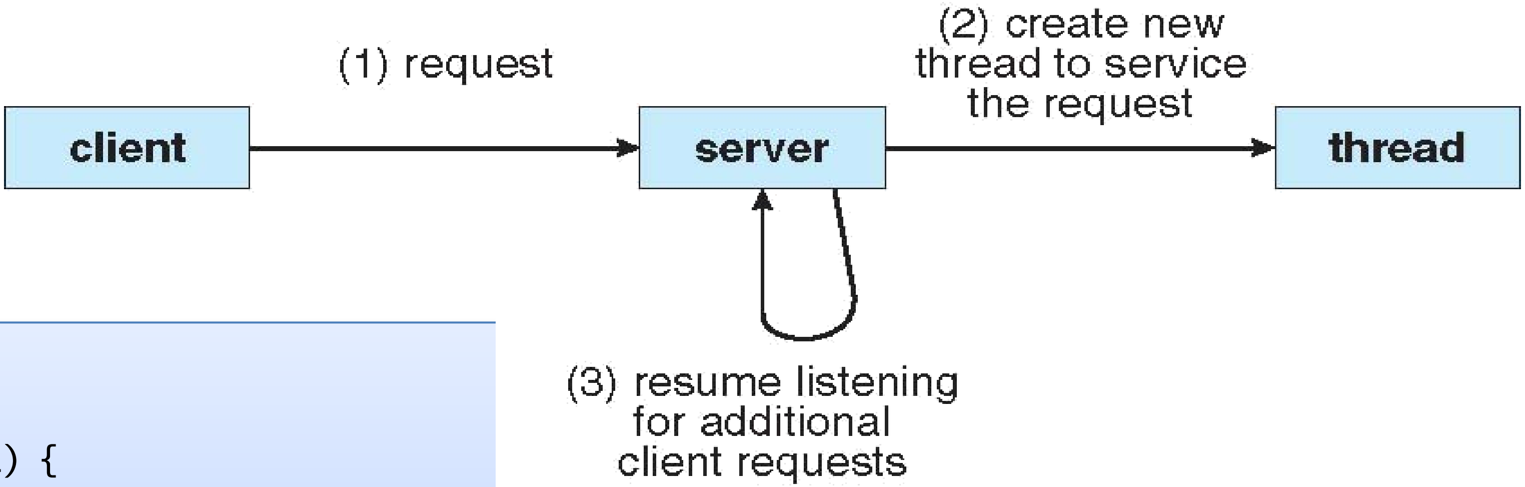
A thread of control (or a thread)

- A sequence of instructions being executed in a program
- Usually consists of
  - A program counter (PC), general registers
  - A stack to keep track of local variables and return addresses
- Threads share the process instructions and most of its data
  - A change in shared data by one thread can be seen by the other threads in the process
- Threads also share most of the OS state of a process

Concurrent Servers: Threads

Using threads

- We can create a new thread for each request


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

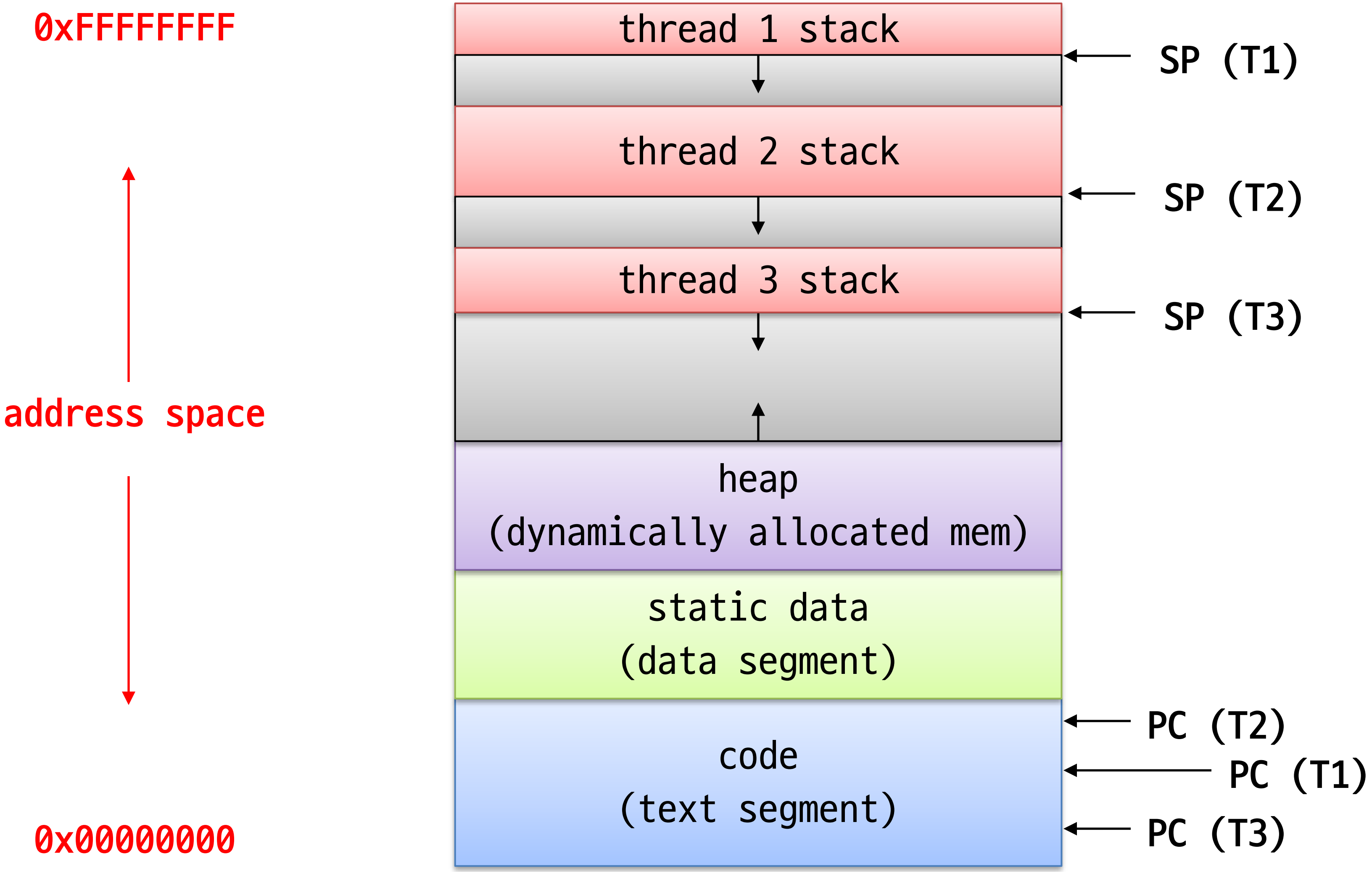
```
webserver ()
{
     while (1) {
          int sock = accept();
          create_thread (handle_request, sock);
     }
}
handle_request (int sock)
{
     /* Process request */
     close (sock);
}
```

Multithreading

Benefits

- Creating concurrency is cheap
  - Time and memory consumption
- Improves program structure
- Higher throughput
  - By overlapping computation with I/O operations
- Better responsiveness (User interface / Server)
  - Can handle concurrent events (e.g., web servers)
- Better resource sharing
- Utilization of multiprocessor architectures
  - Allows building parallel programs

Address Space with Threads



pthreads (1)

Thread creation/termination

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

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

```
int pthread_join   (pthread_t tid,
                    void **thread_return);
```

The Pthreads "hello, world" Program

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

void *threadfunc(void *vargp);

/* thread routine */
void *threadfunc(void *vargp) {
  sleep(1);
  printf("Hello, world!\n");
  return NULL;
}

int main() {
  pthread_t tid;

  pthread_create(&tid, NULL, threadfunc, NULL);
  printf("main\n");
  pthread_join(tid, NULL);
  printf("main2\n");
  sleep(2);
  return 0;
}
```

```
# gcc ex.c -lpthread
# ./a.out
main
Hello, world!
main2
```

Threading Issues (1)

fork() and exec() can be issue

When a thread calls fork()

- Does the new process duplicate all the threads?
- Is the new process single-threaded?

Some UNIX systems support two versions of fork()

- In pthreads,
  - fork() duplicates only a calling thread
- In the Unix international standard,
  - fork() duplicates all parent threads in the child
  - fork1() duplicates only a calling thread

Normally, exec() replaces the entire process

If a thread call exit()?

If the main thread dies(return, exit()) before child threads?