Operating Systems

Complutense University of Madrid

2015-2016

Unit 3.1: Process Management

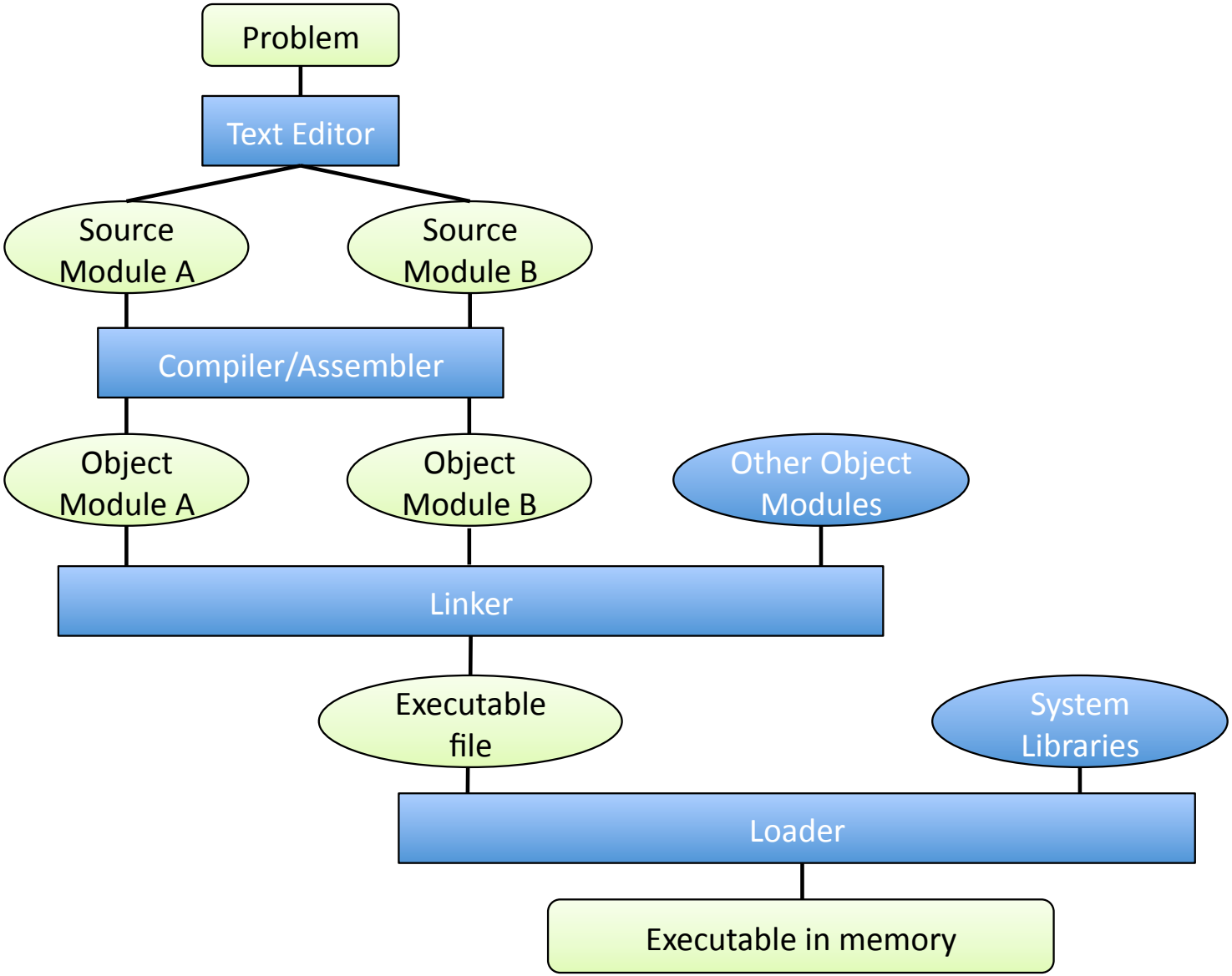
Contents

Contents

Concept of Process

- **Program**: executable file in a non-volatile storage device
- **Process**: program in execution
  - Set of resources that make it possible a program's execution
    - Memory image
    - Open Files
    - Threads
    - ...
- A process may consist of one or several *threads* of execution
  - A thread is the minimal schedulable entity
  - What can be actually mapped to a CPU

Preparing a process's code



Reminder: useful comands

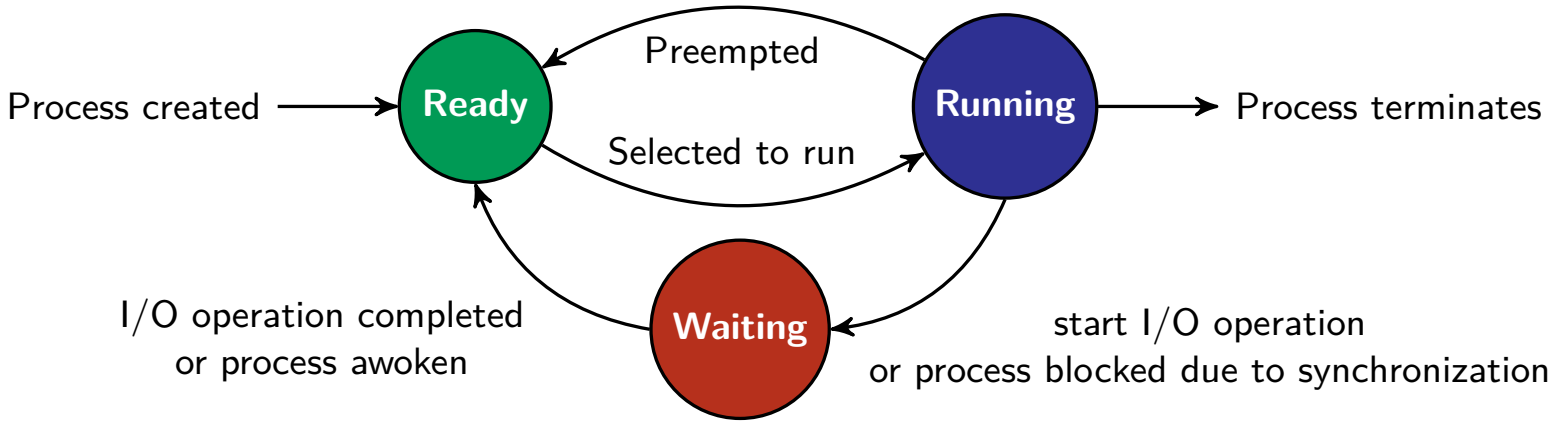
- `gcc`
  - GNU C Compiler
  - Performs all the steps

```
$ gcc --save-temps hello.c -o hello.o
```

- `ldd`
  - Usage: `ldd <executable-file>`
  - Makes it possible to list the dynamic libraries an executable file depends upon
- `nm` / `objdump`
  - Visualize parts of an executable file

Basic process states

1. Running (one per CPU/core)
2. Waiting/Blocked (blocked due to an I/O operation or due to synchronization)
3. Ready to run



- Scheduler: Component of the operating system that decides which process runs on which CPU at any time
- Idle process/task (one per CPU/core)

Process environment

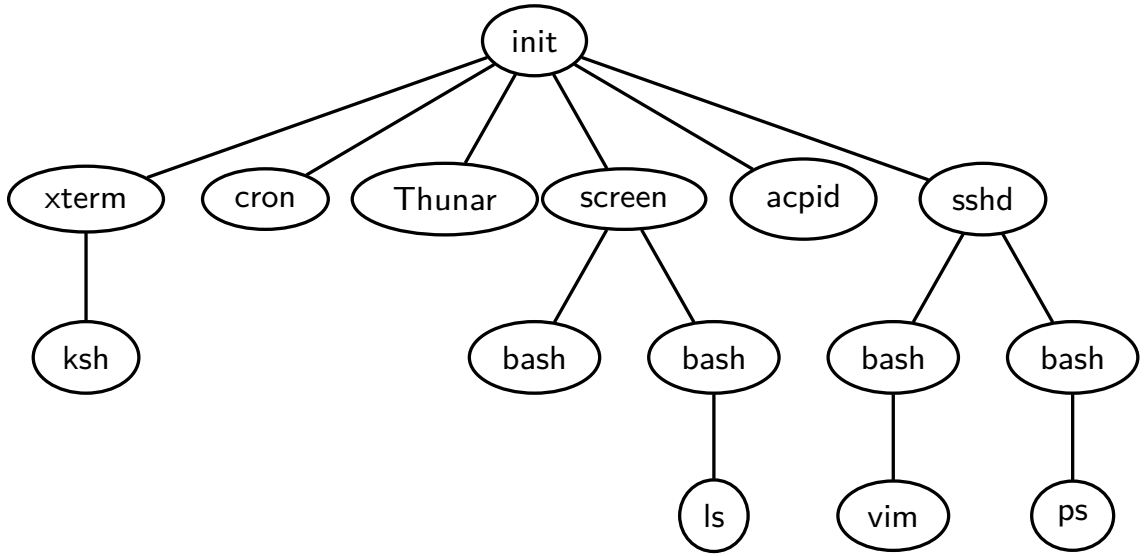
- **Environment:** table of variables that is passed to the process upon creation
  - Each table entry is a (*name*,*value*) pair
  - It can be listed with the `env` command
- Table's initial contents inherited from the parent process
  - Can be changed by means of:
    1. Functions of the std C library: `putenv()`, `getenv()`, `unsetenv()`
    2. Shell commands: `export NAME=value`, `unset NAME`

Example

```
$ env
PATH=/usr/bin:/home/joe/bin
TERM=vt100
HOME=/home/joe
PWD=/home/joe/books/OperatingSystems
TIMEZONE=EDT
...
```

Processes' hierarchy (UNIX)

- On UNIX, processes are created by cloning another process
  - fork()
- Process family:
  - Child process
  - Parent process
  - Sibling process
  - ...
- Process Life-cycle
  - Creation
  - Execution
  - Death/Termination



Watching active processes

- `ps`
  - Allows to retrieve the information on all processes in the system
- `top` (*table of processes*)
  - Shows a table with the information on all processes
  - Information is refreshed periodically
  - Interactive features (e.g., send signal to a process)
- `pstree`
  - Displays a text-based representation of the process tree hierarchy

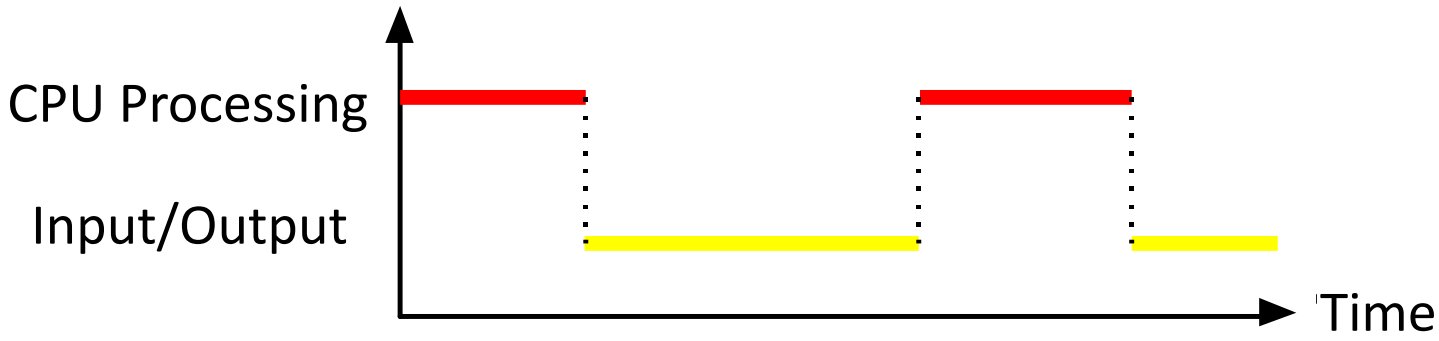
Users and Groups

- Every process has an associated *owner* user
- User: person with authorization to use the system
  - During the login process a user is identified with:
    - User name
    - Password
  - Internally, the OS assigns each user a "UID" (*User identification*)
- Superuser (*root*)
  - Has all permissions
  - Is in charge of the system's administration
- User groups
  - Users must belong to at least one group
  - Groups make system administration simpler

Contents

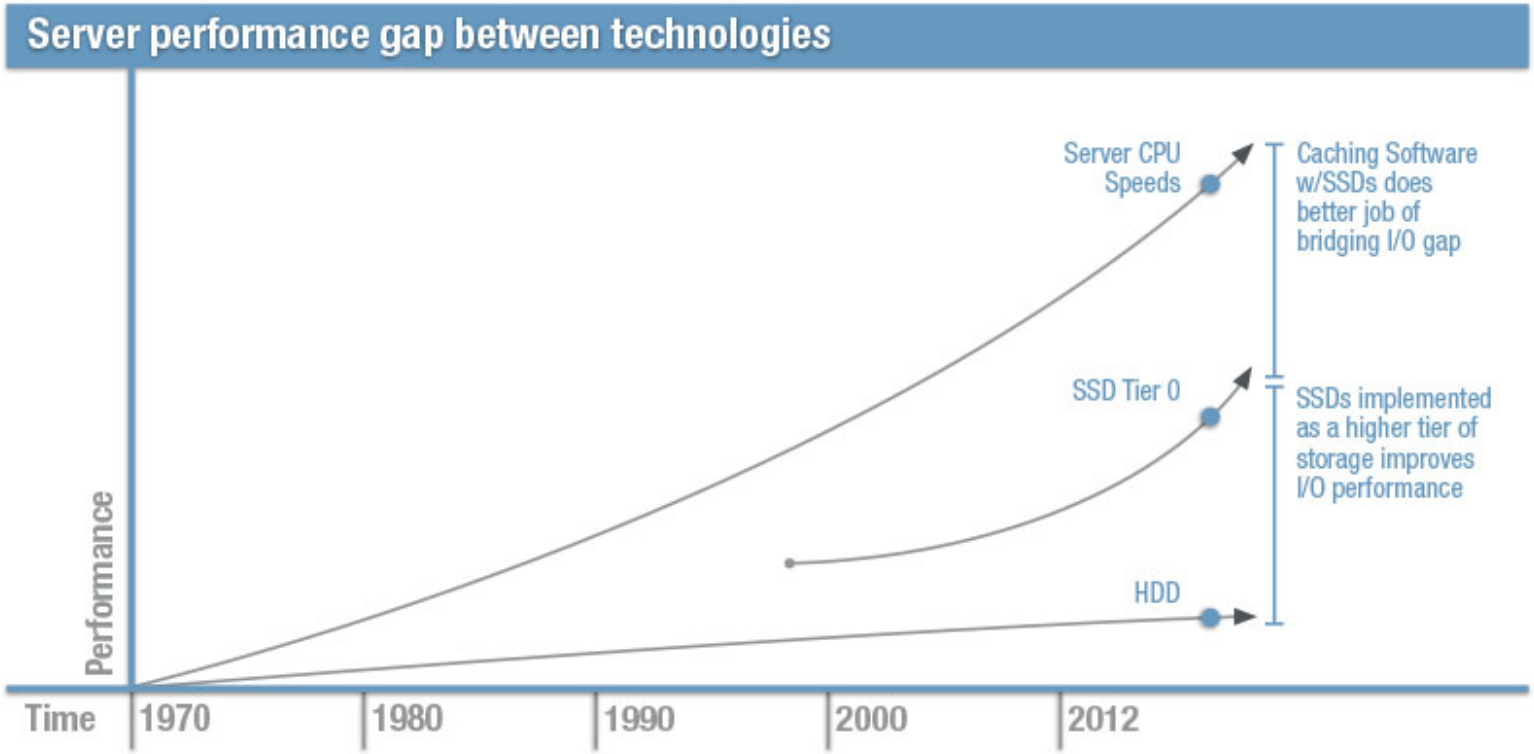
Motivation of Multitasking

- In batch OSes (50s/60s), processes were executed serially (one after another)
  - System process queue
- When a running process initiated an I/O operation, the CPU became idle
  - Use of programmed I/O (busy waiting)



Motivation of Multitasking

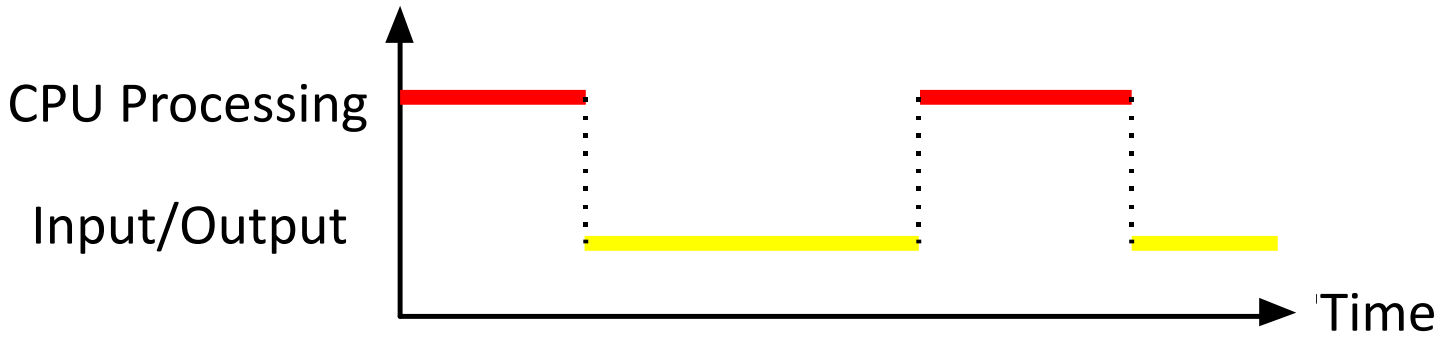
- Since the emergence of the CMOS technology until today, the gap between the CPU speed and the speed of I/O devices has increased dramatically
  - Programmed I/O becomes even more inneficient



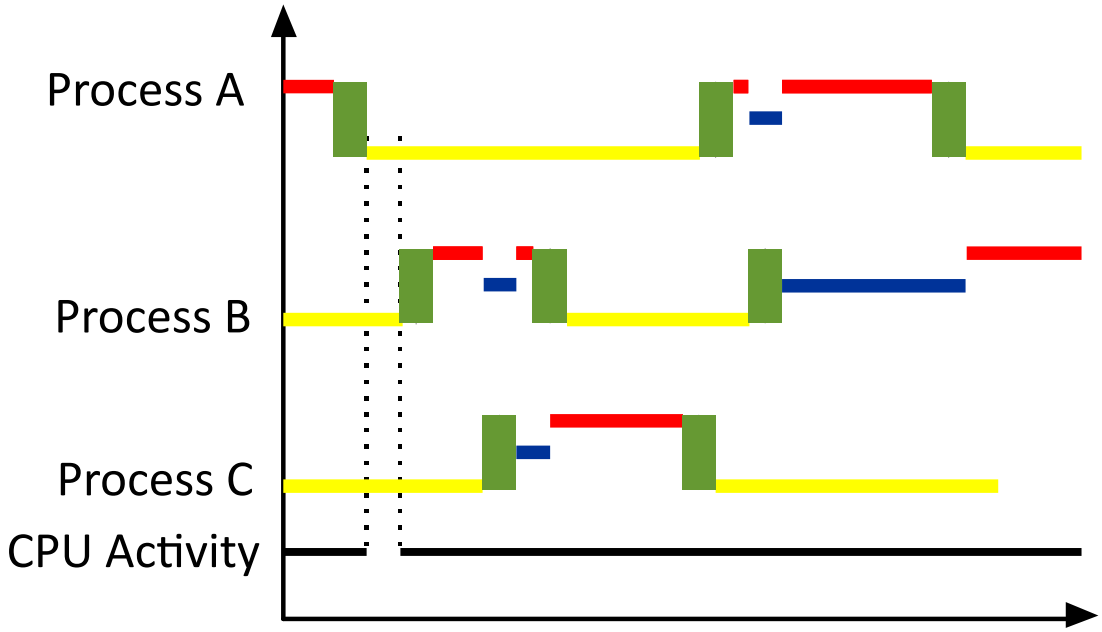
Source: RTC magazine

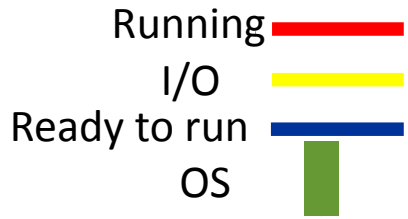
Basics of Multitasking

- Exploits the fact that processes go through I/O phases and CPU processing phases
- Real parallelism between I/O processing and CPU processing
  - Many processes can perform I/O operations in parallel
- Memory stores the memory image of multiple processes



Process execution with multitasking





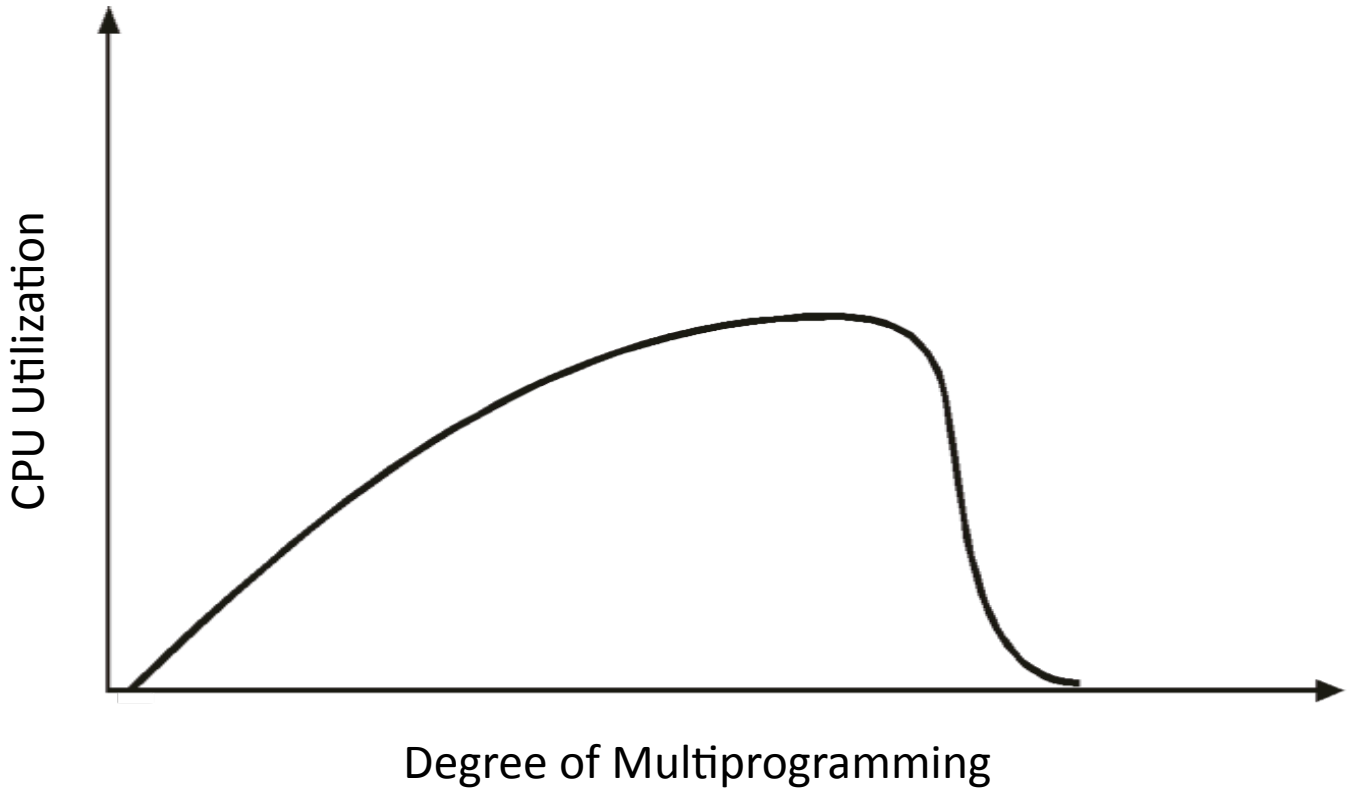
- Idle process

Benefits of Multitasking

- Simplifies programming, by enabling us to divide programs into multiple processes (modularity)
- Enables processes from multiple users to be serviced efficiently
- Optimizes CPU utilization

Degree of multiprogramming

- **Degree of multiprogramming**: maximum number of processes that a system can accommodate efficiently
- For systems with virtual memory:



Contents

Process information

- **Processor state**: value of the various architectural registers at a given point
- **Memory Image**: contents of the memory regions that store the process's data and code
- **Process Control Block (PCB), Process Descriptor or Task Structure**:
  - Process state (running, blocked, ...)
  - Temporary storage for processor state
    - Up-to-date values only when the process is not currently running
  - Process identifiers: PID, PPID, UID, etc.
  - Process priority
  - Memory segments/regions (Process address space)
  - Open files
  - Timers
  - Signal-related information
  - ...

Processor state

- Snapshot of the contents of the various processor registers exposed to software
  - General-purpose registers
  - Program counter
  - Stack pointer
  - State registers (flags, ...)
  - Special registers
- When a process blocks or is evicted from the CPU the processor state is saved in the PCB
  - A "snapshot" is not created periodically while the process runs, so stale data is stored in the PCB in the meantime

Process memory image

- A process's memory image comprises a set of memory regions that the process is authorized to use
- The memory image, depending on the computer capabilities, can be associated either with physical memory or with virtual memory
- If a process accesses a memory region outside of its address space, the memory-management HW raises an exception that the OS will handle
  - Typically, the offending process is killed

Information in the PCB (I)

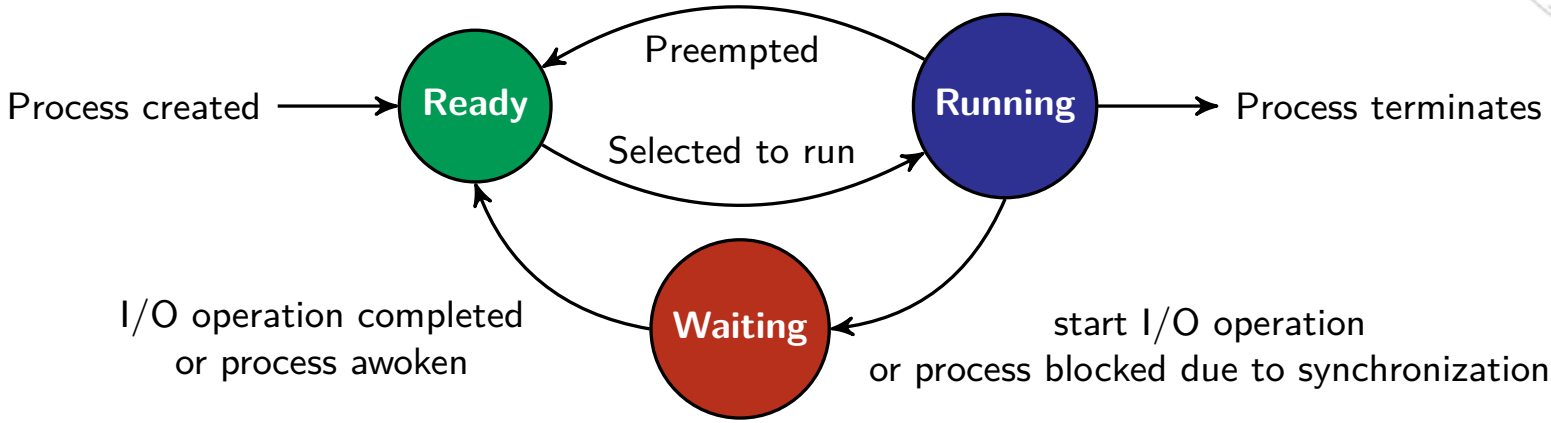
- **Identification**
  - Process ID (PID), PID of the parent process (PPID)
  - IDs of the owner user (UID) and group (GID) (real UID/GID)
  - IDs of the effective user and effective group (effective UID/GID)
- **Processor state**
- **Control information**
  - Scheduling-related information (state, priority, CPU accounting)
  - Descriptor of the process memory image
  - Resources associated with the process (open files, syncronization resources, pending signals,...)
  - Various pointers to make it possible to insert the PCB in various global OS data structures (task list, process tree, scheduler queues – run queues–, wait queues, ...)

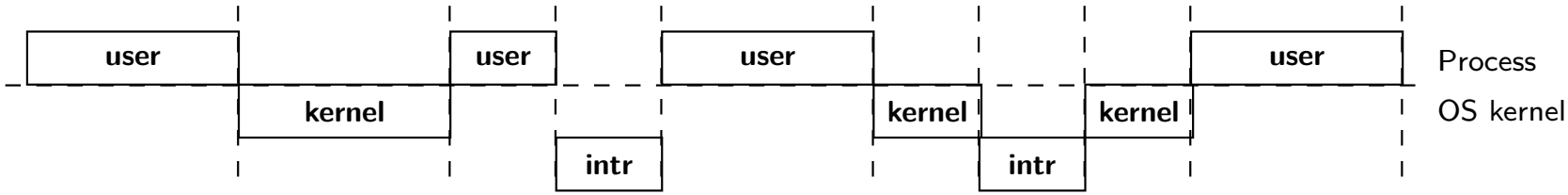
Information in the PCB (II)

- Several resources in the PCB may be shared among multiple processes or tasks (multiple PCBs)
  - Page tables (virtual memory)
  - Open files (inherited after forking a new process)
  - ...
- For these resources a pointer to a structure is maintained in the BCP rather than the structure itself
  - Processes sharing the same resource hold the same pointer (address) in their BCPs
  - Typically a reference counter is needed to keep track of the number of processes using the resource
    - The structure will be freed up when ref counter = 0

Contents

Process states





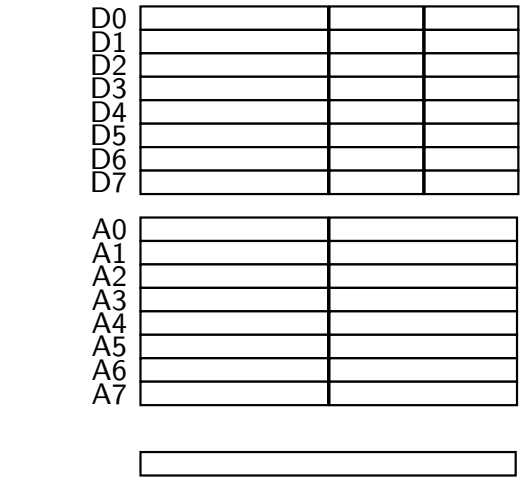
Changes in the processor mode

- When an interrupt/exception ocurrs while a process runs in user mode, the processor switches to kernel mode

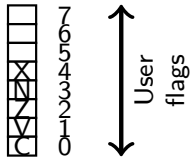
- In addition...
  - The processor state is saved in the kernel stack of the process
  - The ISR/ESR (Interrupt/Exception Service Routine) is executed (OS code)
- Upon completion of the ISR/ESR, if the process is still runnable:
  - The OS restores the processor state with the values stored in the process kernel stack
  - Executes a special HW instruction (RETI) to return from interrupt. Two simultaneous actions:
    1. Switches back to user mode by writing into the state register
    2. Loads the "right" value in the PC, so that the process may pick up from where it left off in user mode

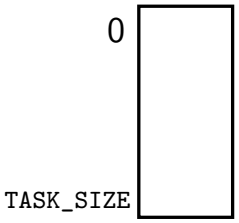
User mode vs. kernel mode

User mode



State register

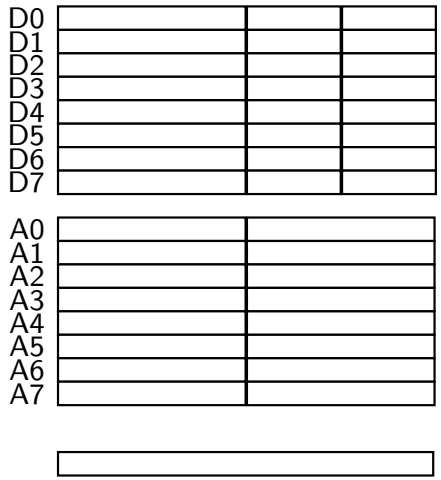




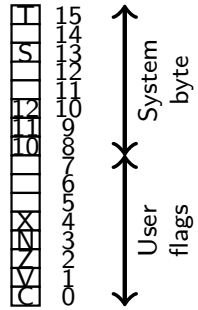
Virtual address space

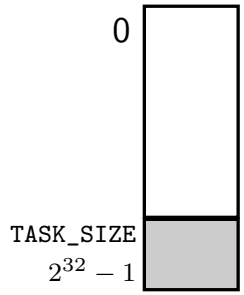
Instruction set

Kernel mode

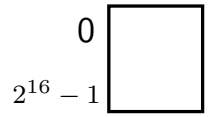


State register





Virtual address space



I/O address space

Instruction set

Context switch

- A context switch is the set actions performed by the OS to change the process currently running on a CPU
- Steps:
  1. Save the context of the outgoing process (register values) in the PCB
  2. If the outgoing process is still runnable (preemption) change its state (running $\rightarrow$ ready)
  3. Change the current address space (HW specific)
     - Page tables + Memory segments that the incoming process can use
     - On x86. CR3 (start address of the page directory) + LDTR (address of Local Descriptor Table)
     - In some architectures, flush the TLB
  4. Change the state of the incoming process (ready $\rightarrow$ running)
  5. Restore the context of the incoming process (register values) stored in the PCB, and return to user space

Context switch

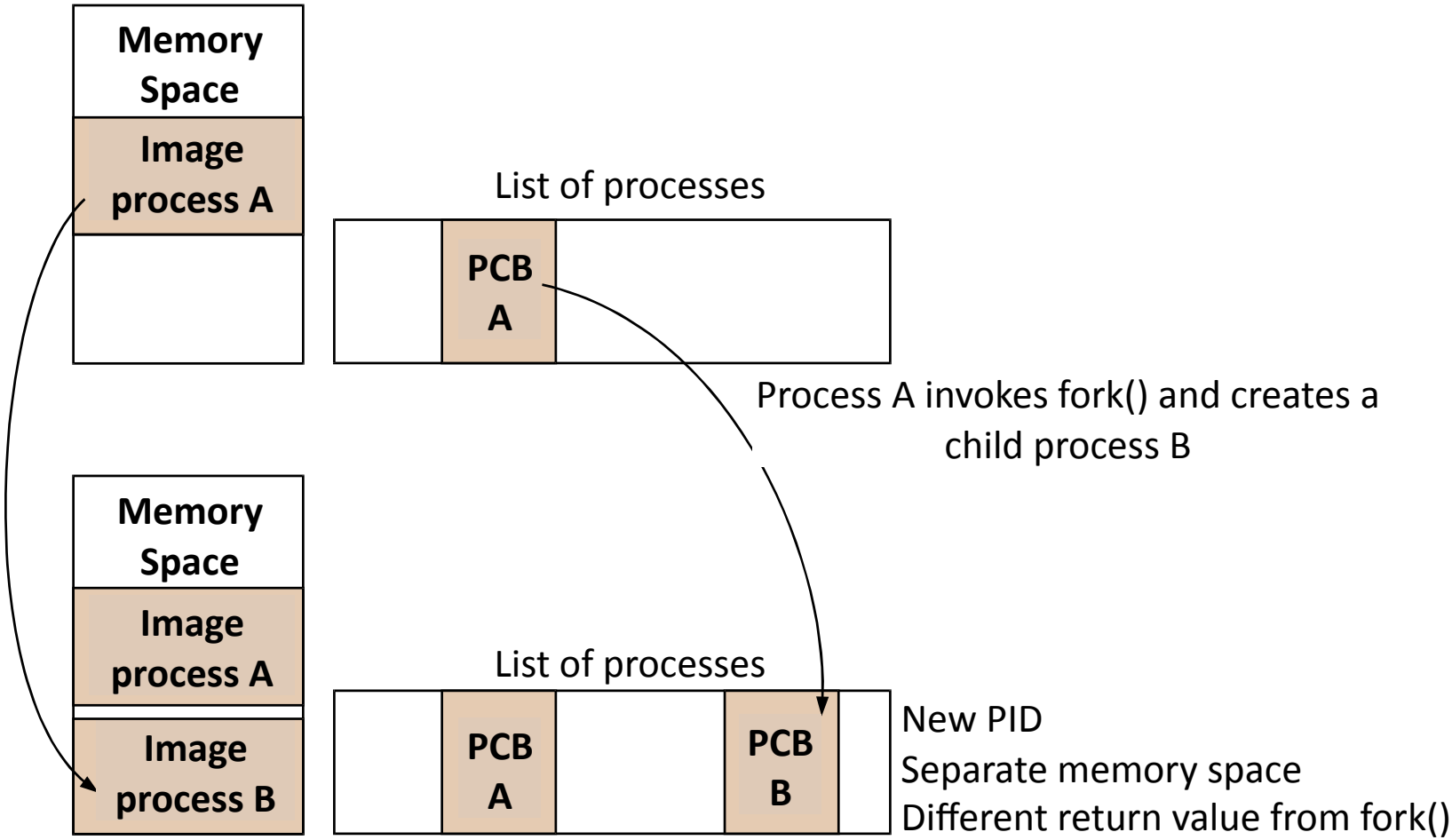
- A context switch can be a rather costly operation
- A change in the processor mode does not always lead to a context switch
  - If the current process is still runnable after processing the interrupt/exception and the scheduler does not decide to preempt the process no context switch will take place

POSIX services for processes

- Process identification
- Manipulating/Accessing a process's environment
- Creating processes
- Changing the program (executable) associated with a process
- Waiting for a process to finish
- Terminate (kill) a process
- Retrieve information about processes

POSIX services: `fork()`

- Creates a process by cloning another process



POSIX services: `fork()`

Service:

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

Return:

- On success it returns twice, once in the parent process and once in the child
- Returns 0 on the child process and the child's PID in the parent process
- On failure, it returns just once (return value=-1)

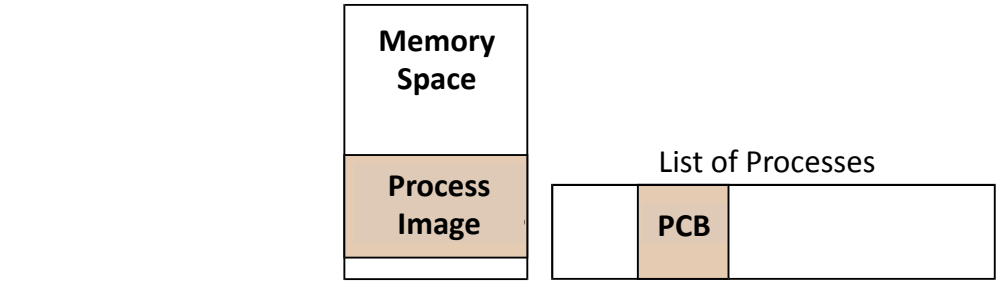
```
void main()
{
    pid_t pid;
    ..
    pid = fork();
    if (pid == 0) {
        /* Child process */
        ...
    } else if (pid>0){
        /* Parent process */
        ...
    } else{
        /* Error path */
    ...
    }
...
}
```

Description:

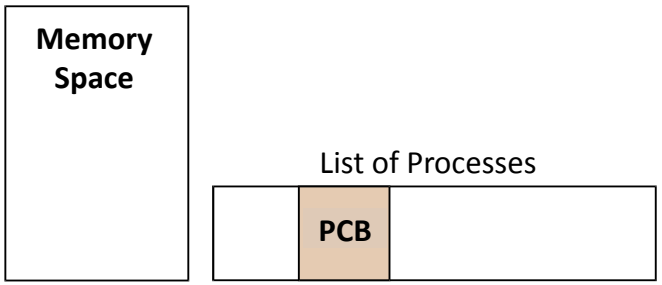
- Creates a child process that runs the same program than the parent
- The child inherits open files
- Child process features one thread of execution

POSIX services: `exec()`

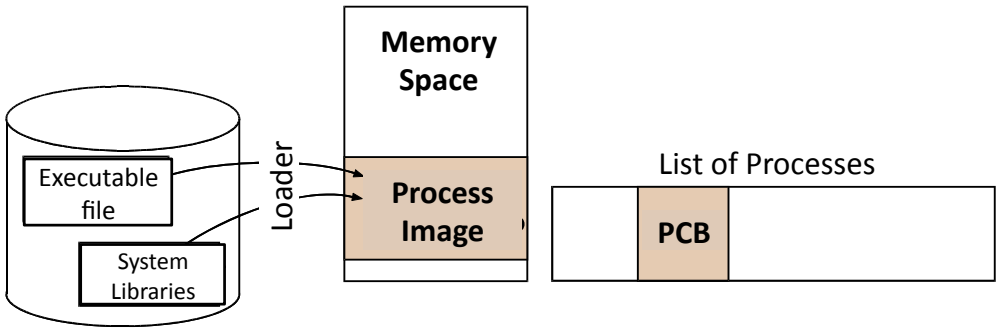
- Changes the program that a process runs



The process invokes exec()



The memory image is destroyed
The memory address space descriptor is reset
The process's PID is preserved



The OS builds a new memory image
The program start address is loaded in the PC
Open files (descriptors) are preserved

POSIX services: `exec()`

Available functions

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
```

Arguments:

- `path`, `file`: path name of the executable file
- `arg`: program arguments

Description:

- On success, **it does not return**; upon failure it returns -1
- The same process executes another program
- Open files remain open
- Signals with the default action remain as is; signals with a handler will now take the default action.

POSIX services: exec() examples

Executing ls -l

- Executable is located under /bin
  - Run command "which ls" to obtain full path
  - We assume that /bin is in the PATH (echo $PATH)

```
/* First approach */
execl("/bin/ls","/bin/ls","-l",NULL);

/* Second approach */
execlp("ls","ls","-l",NULL);

/* Third approach */
char* arguments[]={"ls","-l",NULL};
execvp(arguments[0],arguments);
```

Terminating a process: `exit()`

Service:

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

Arguments:

- Exit status code to be delivered to the parent process

Description:

- Terminates the process's execution
- All open file descriptors are automatically closed
- All resources associated with the process are freed up

Waiting a process's termination: `wait()`

Service:

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

Arguments:

- `status`: output parameter. Exit status code of the child process

Description:

- Returns the identifier of the child process; upon failure `wait()` returns -1
- Enables a parent process to wait until any child process terminates. When a child terminates, the status code as well as the child's PID is returned by `wait()`.

Example

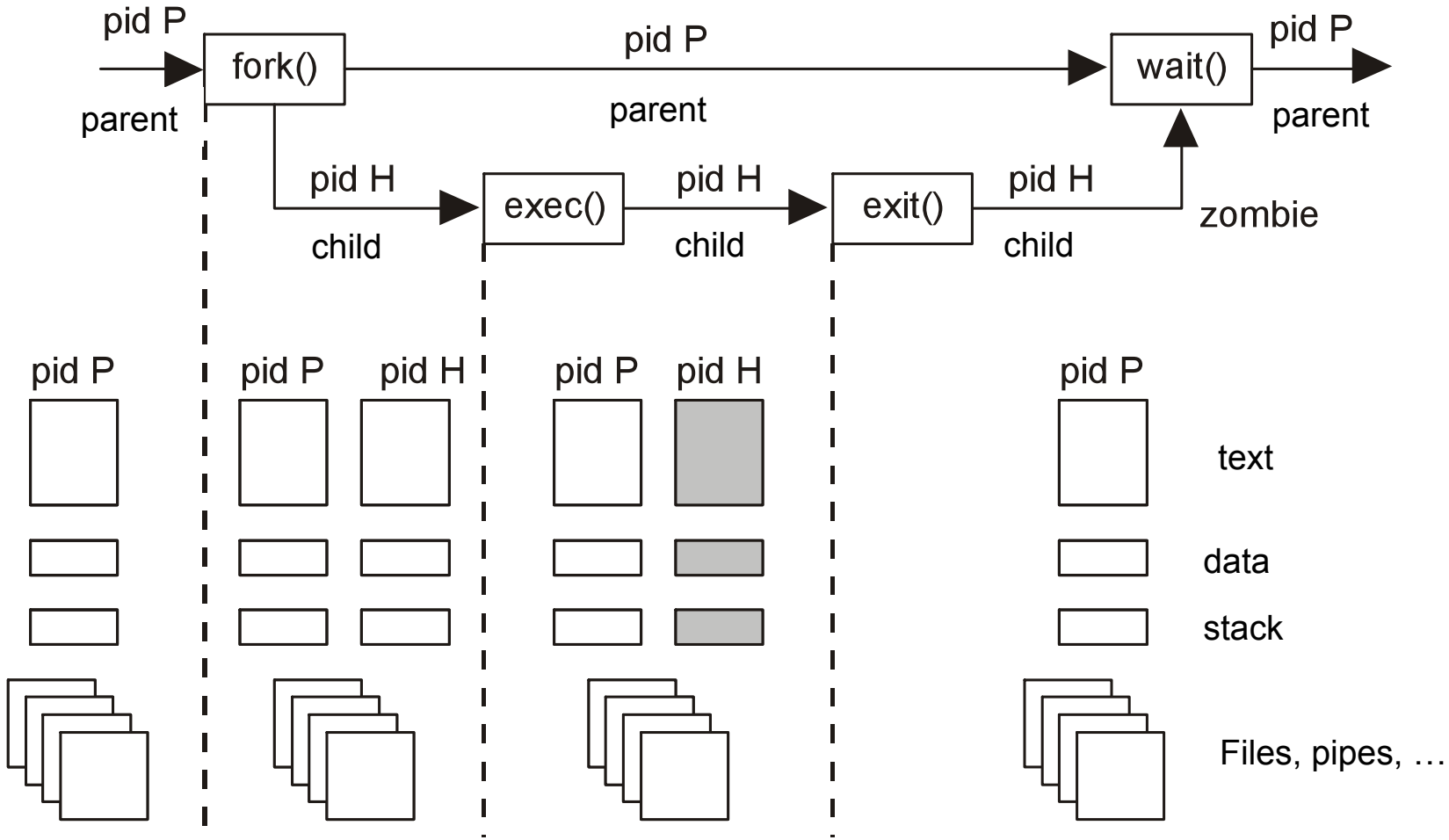
```
void main(){
   int a=0;
    int status;
    pid_t pid;

    pid = fork();
    if (pid == 0) {
    /* child process */
        a++;
    } else if (pid>0){
    /* parent process */
        wait(&status);
        a=a+2;
     } else{
        fprintf(stderr,"Can't fork()\n");
        exit(1);
     }

    printf("My PID is %d, a=%d\n",getpid(),a);
    exit(0);
}
```

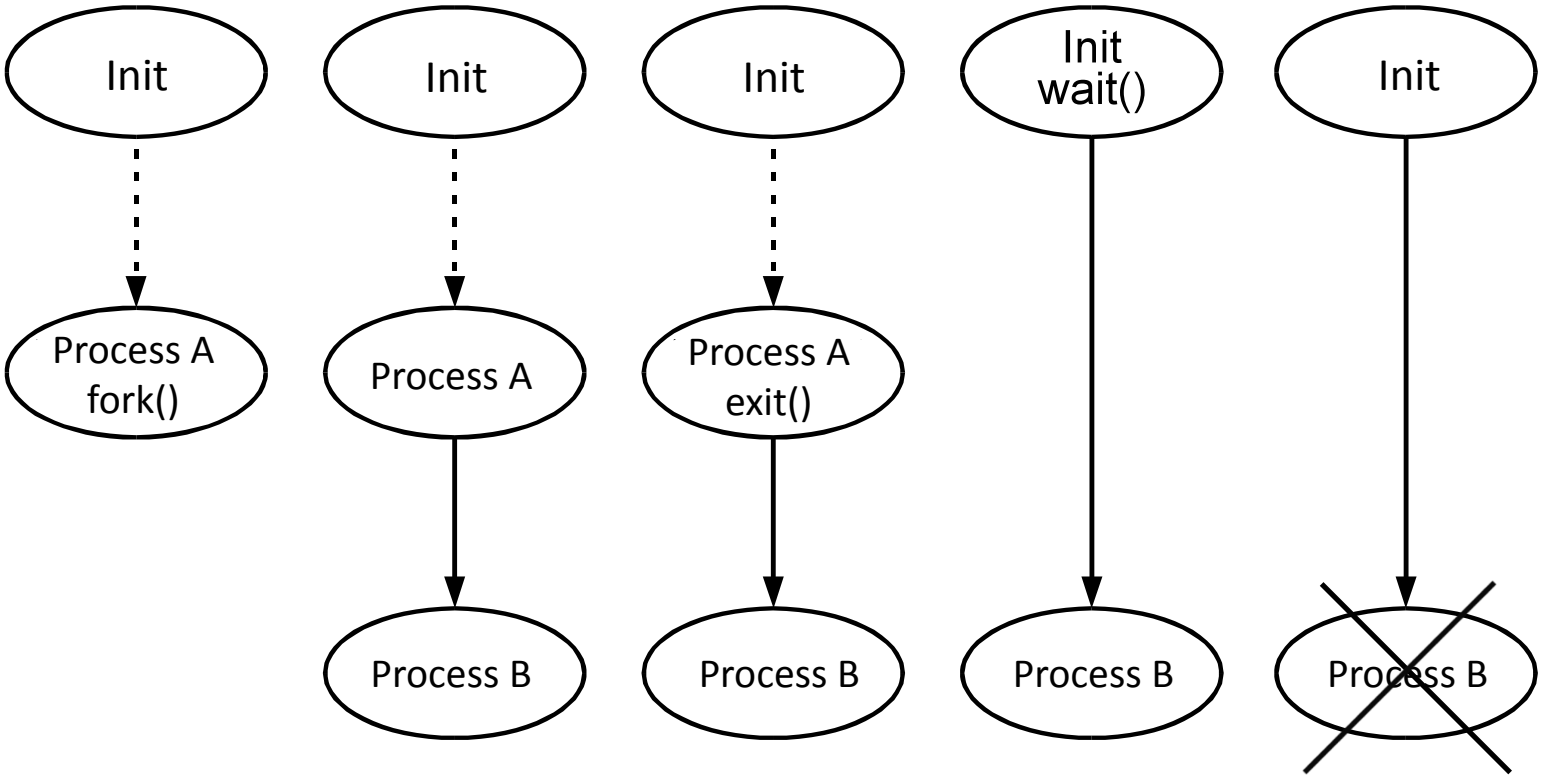
- What will the child process print with `printf()`? 1
- And the parent process? 2
- Would it be the same if a was declared as a global variable instead?

Typical usage of services



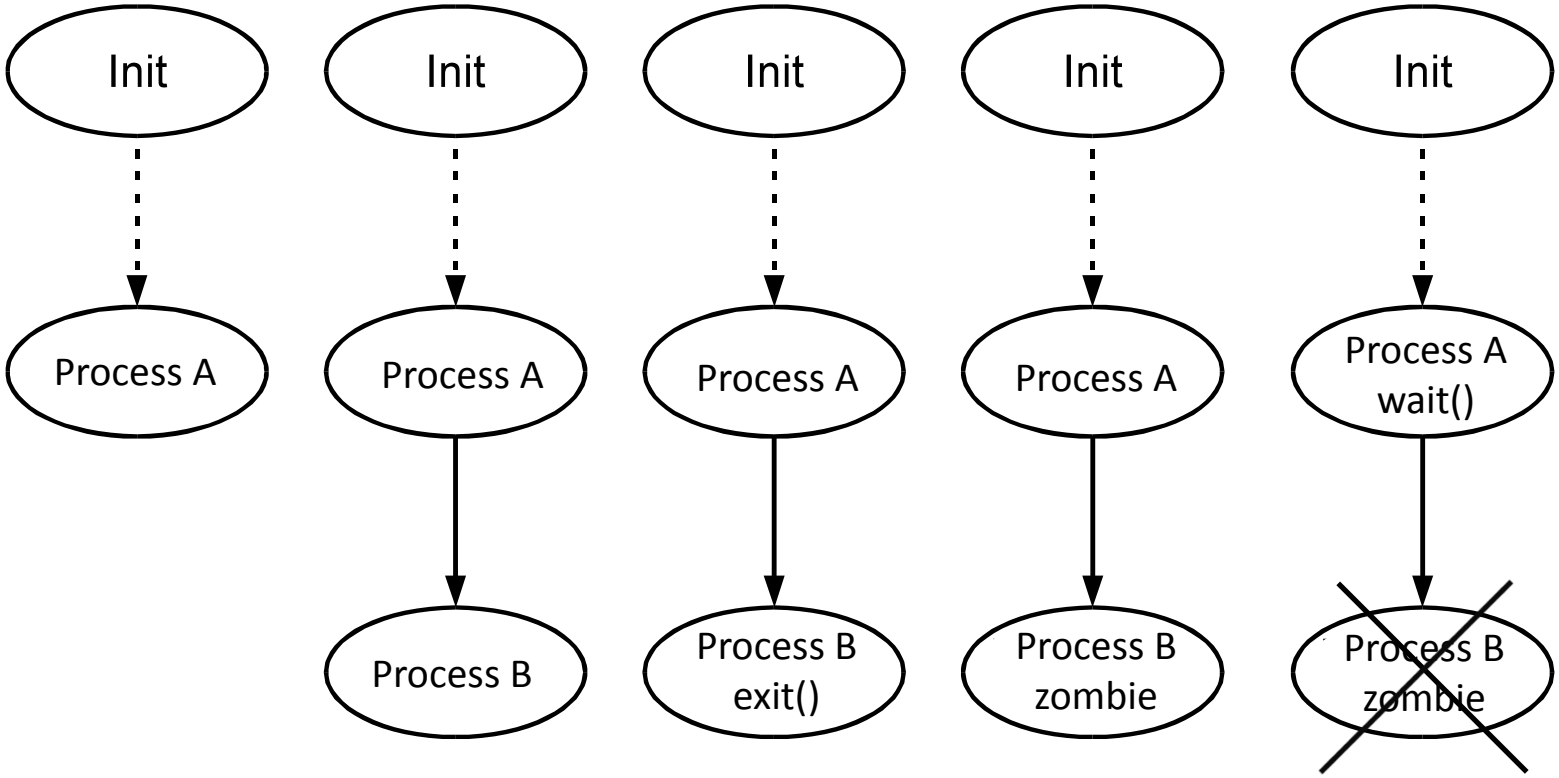
Process life-cycle (I)

- The parent process dies → INIT accepts the children



Process life-cycle (II)

- Zombie: the child process dies but the parent does not invoke `wait()`



Example program

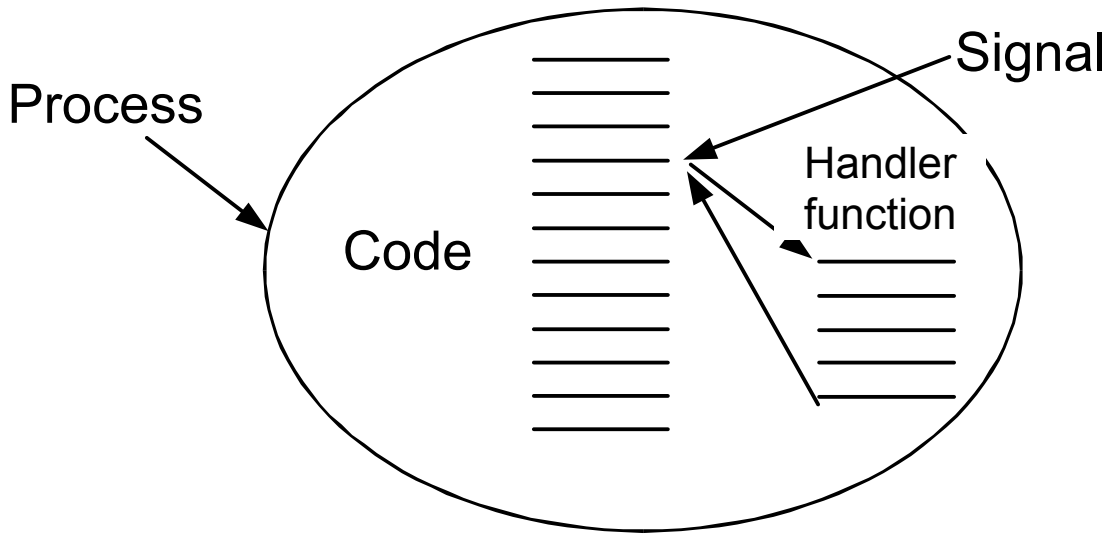
```c
#include <sys/types.h>
#include <stdio.h>

/* program that executes the "ls -l" command */
void main() {
    pid_t pid;
    int status;
    pid = fork();
    if (pid == 0) { /* child process */
        execlp("ls","ls","-l",NULL);
        exit(-1);
    } else /* parent process */
        while (pid != wait(&status));
        exit(0);
    }
}
```

Contents

Concept of signal

- Signals are just like interrupts to processes
- Sending signals
  - Process → Process (with the same `uid`) with `kill()` or via the `kill` command
  - OS → Process



Signals

- There are many types of signals:
  - `SIGILL` illegal instruction
  - `SIGALRM` the process timer fired
  - `SIGKILL` kills the process
- The OS is responsible for delivering the signal to the process
  - The process must be ready to deal with the signal
    - By specifying installing a handler function with `sigaction()`
    - By masking the signal with `sigprogmask()`
  - If the process is not ready $\rightarrow$ default action
    - For most signals the default action is to terminate the process immediately
    - Some signals are just ignored by default
- `pause()`: the process blocks until receiving a signal

Sending signals: example

Terminal 1

```
osuser@debian:~$ sleep 50
^C
osuser@debian:~$ sleep 50

osuser@debian:~$ ^C
osuser@debian:~$ ^C
osuser@debian:~$ echo $$
5711
osuser@debian:~$ ^C
osuser@debian:~$ ^C
osuser@debian:~$
```

Terminal 2

```
osuser@debian:~$ ps -ef | grep sleep
osuser    5756  5711  0 09:19 pts/3    00:00:00 sleep 50
osuser    5758  5667  0 09:19 pts/2    00:00:00 grep sleep
osuser@debian:~$ kill -s SIGINT 5756
osuser@debian:~$ kill -s SIGINT 5711
osuser@debian:~$ kill -s SIGINT 5711
```

The $$ built-in variable in BASH stores the PID of the shell process

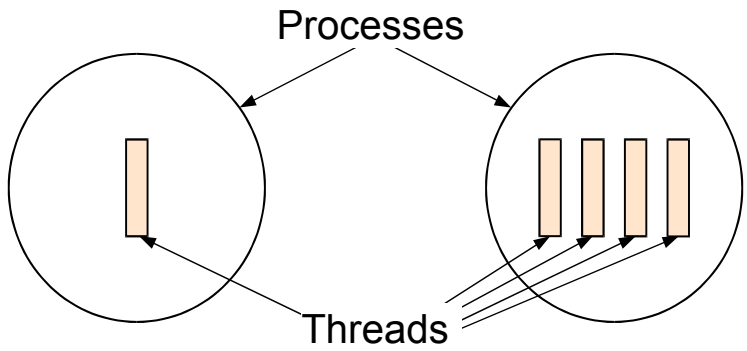
POSIX services for signals

- `int kill(pid_t pid, int sig);`
  - Sends the signal `sig` to process with PID=`pid`.
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - Enables to indicate which action must be performed (`act`) when the process receives the `signum` signal. The previous installed action can be optionally stored in `oldact`
- `int pause(void);`
  - Blocks the process until it receives a signal
- `unsigned int alarm(unsigned int seconds);`
  - Enables to program the per-process timer to send the SIGALRM signal to the process after the specified amount of time in seconds.
- `int sigprocmask(int how, const sigset_t *set, sigset_t *oset);`
  - Used to examine/modify the process's signal masks

Contents

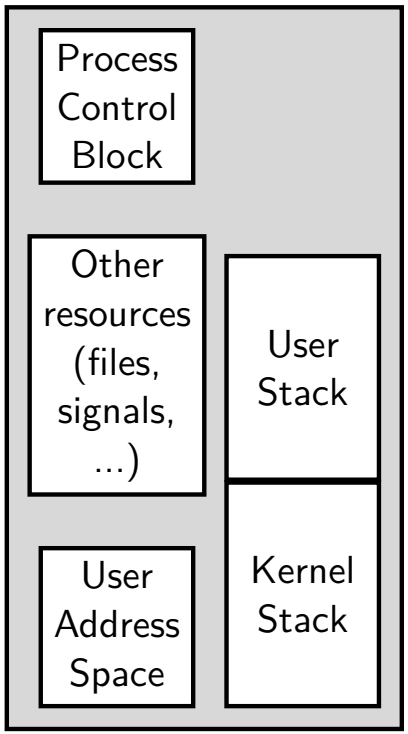
Threads

- Per-thread data
  - Program counter, registers
  - Stack
  - State (running, ready or waiting)
  - Thread Control Block (TCB)
- Per-process data
  - Memory address space
  - Global variables
  - Open Files
  - Child processes
  - Timers
  - Semaphores and signals

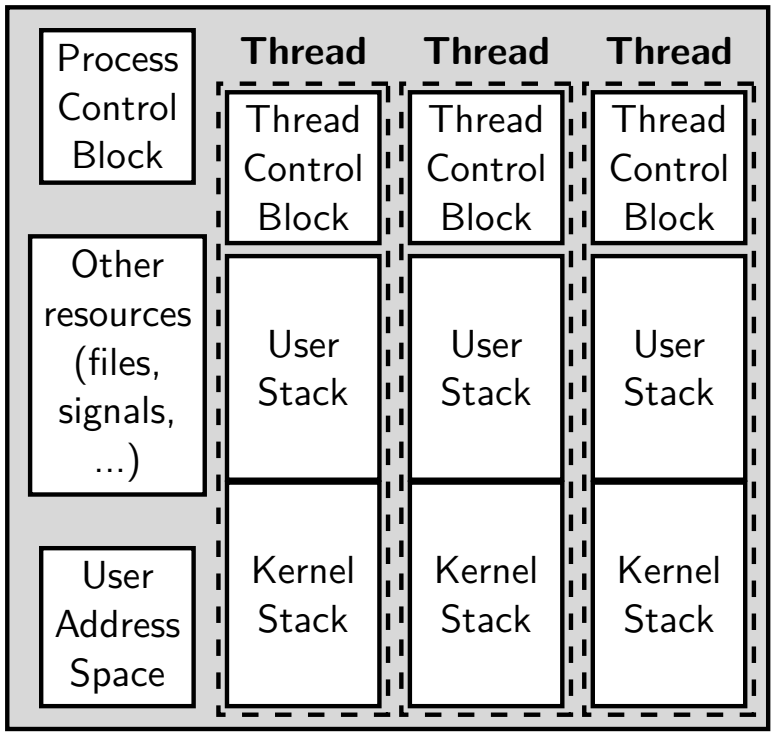


Single-threaded vs. Multithreaded model

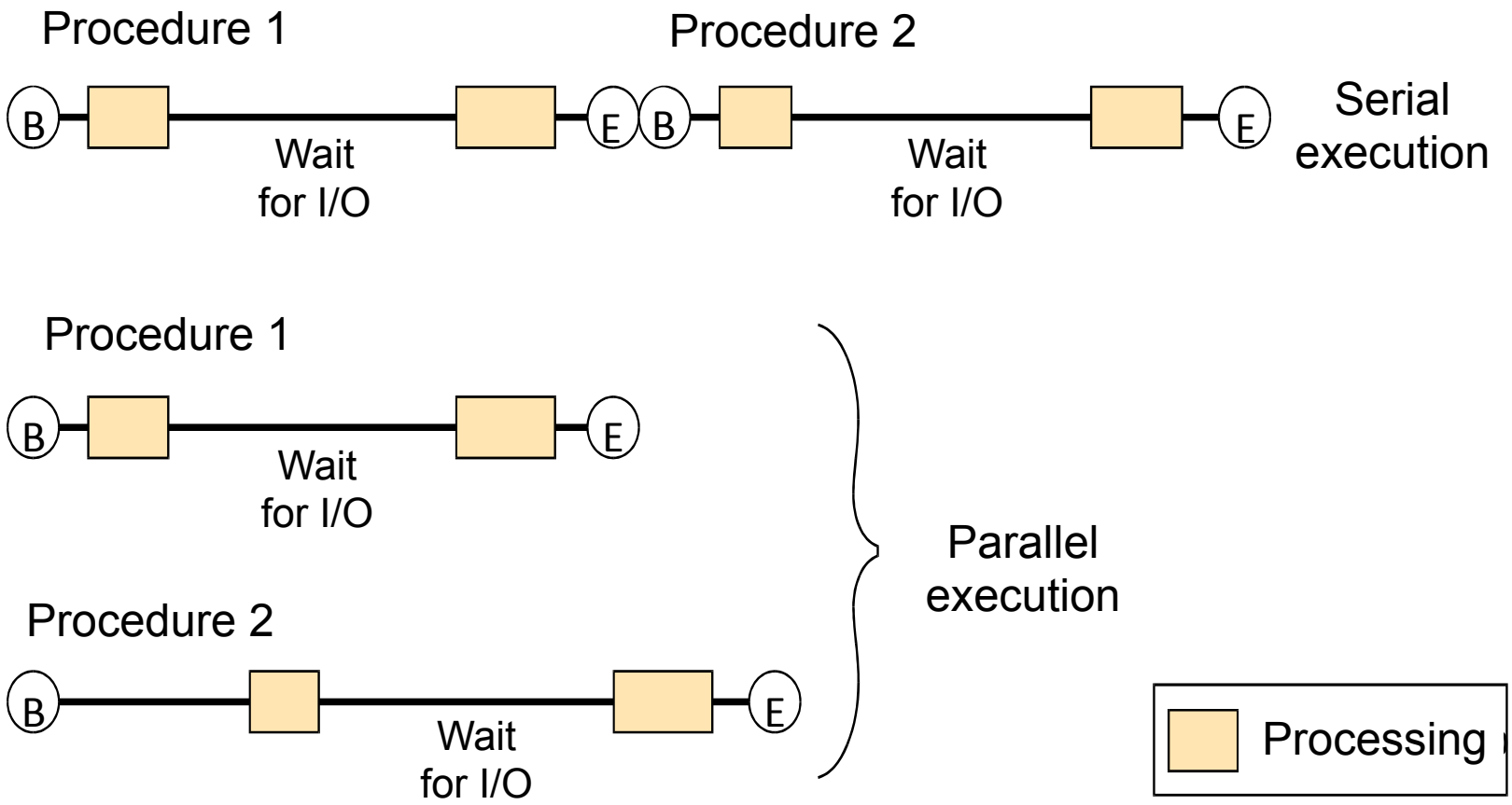
Single-threaded Process Model



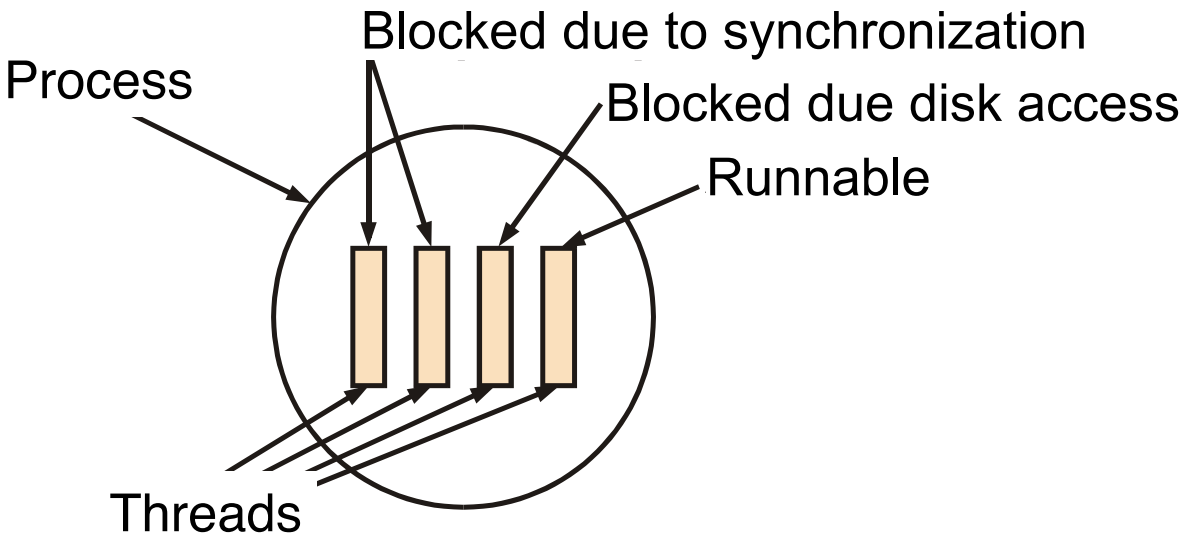
Multithreaded Process Model



Parallelization using threads



Thread states



Advantages threads vs. processes

- Operations related to creation, destruction, scheduling and synchronization are typically more costly for processes than for threads
  - Up to an order of magnitude
- A context switch between two threads of the same process is much more efficient that switching between two threads belonging to different processes
  - In the former case, there is no need to switch the "active" address space

Design with threads

- Makes it possible to run several tasks in parallel
- Enables to break down a task into several sub tasks
- Speeds up the execution of a given program as long as we have multiple CPUs/cores
- Paralellism
- Blocking system calls
- Concurrent programming (shared memory machines)
  - Shared variables/data structures
  - Reentrant functions
    - Picture several simultaneous invocations of the same function (not necessarily coordinated)
  - Synchronization mechanisms among threads
    - mutexes, semaphores,...
  - Simplicity vs. mutual excusion vs. scalability

Alternatives to multithreading

- Single-threaded process
  - No parallelism
    - Blocking system calls
  - Parallelism managed by the programmer
    - Asynchronous (Non-blocking) system calls
- Multiple conventional processes cooperating
  - Allows paralellism
  - Processes do not share variables by default
    - Necessary an OS-provided inter-process communication mechanism
  - Higher communication/synchronization overhead

POSIX Threads API

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*func)(void *), void *arg)`
  - Creates a thread that runs `func` with argument `arg` and attributes `attr`.
  - Attributes enable to specify: stack size, priority, scheduling policy, etc.
  - Several function exist to modify attributes
- `int pthread_join(pthread_t thid, void **value)`
  - Blocks the caller thread until the thread with `thid` identifier terminates.
  - Returns the termination status of the thread.
- `int pthread_exit(void *value)`
  - Allows a thread to terminate the execution immediately, and return its termination status.
- `pthread_t pthread_self(void)`
  - Returns the identifier (numeric descriptor) of the calling thread.

POSIX Threads API (Cont.)

- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`
  - Enables to specify the termination behavior of a thread.
  - If `detachstate = PTHREAD_CREATE_DETACHED`, the thread resources will be freed up as soon as the thread terminates.
  - If `detachstate = PTHREAD_CREATE_JOINABLE`, the thread resources will not be freed up as soon as the thread terminates. Instead another thread must invoke `pthread_join()` on this thread to ensure that its resources are freed up.

Example program

```
#include <stdio.h>
#include <pthread.h>
#define MAX_THREADS 10

void* func(void* arg) {
    printf("Thread %d \n", pthread_self());
    pthread_exit(0);
}

int main(void) {
    int j;
    pthread_attr_t attr;
    pthread_t thid[MAX_THREADS];

    pthread_attr_init(&attr);

    for(j = 0; j < MAX_THREADS; j ++)
        pthread_create(&thid[j], &attr, func, NULL);

    for(j = 0; j < MAX_THREADS; j ++)
        pthread_join(thid[j], NULL);

    return 0;
}
```