COMP1001

Computer Systems

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk

Website:
https://www.plymouth.ac.uk/staff/vasilios-kelefouras

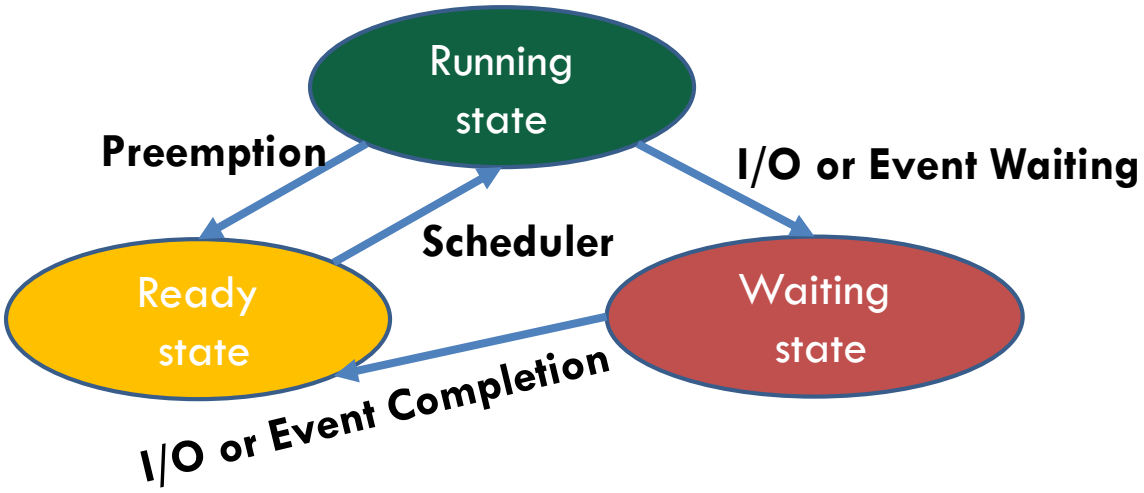
School of Computing
(University of Plymouth)

Outline

- Processes States
- Context Switch
- System Calls
- Processes
- Cloning a process
- Threads
- Pthreads

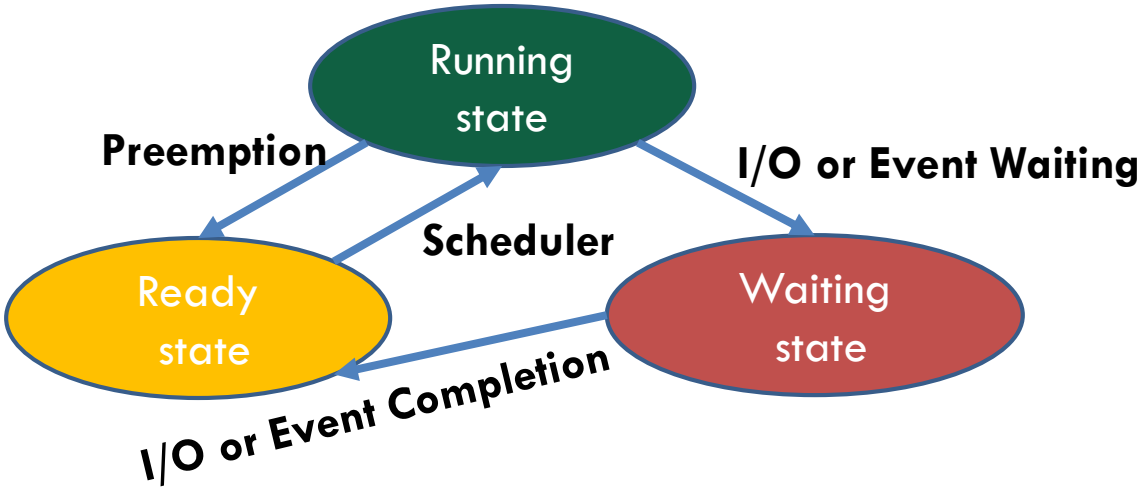
Process States (1)
Simplistic point of view

- At any point in time, a **process** can be in one of **several states:**
  - **Running State:** The process is running using CPU and memory resources
  - **Ready State:** A process is ready to run, but it is not running
  - **Waiting State:** The process cannot run because it is waiting for some event to occur (or data)



Process States (2)
Simplistic point of view

- On a single core CPU, only one process can run at a time, but many processes may be ready for execution or may be blocked
- The OS has a list of the ready and another for the blocked processes
- The scheduler is responsible for choosing the next process to run
- If a running process exceeds a running time limit, then the OS will stop (**pre-empt**) the process and put it back in the ready list. Then, another process will be put to the running state
  - This is called **preemptive multitasking**

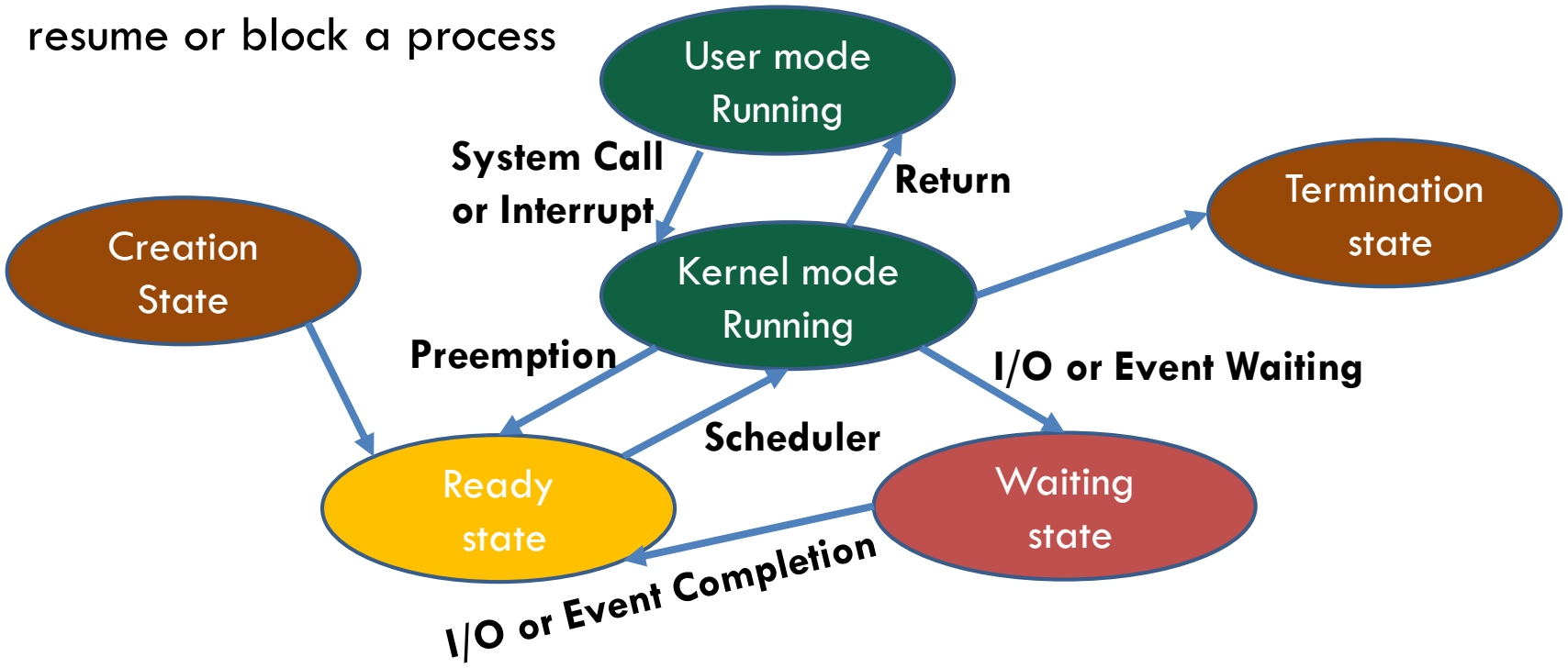


Context Switch

- The **scheduler** is responsible for deciding **which process to run** and for **saving and restoring the state of a process** as it gets stopped and when it runs again
- The context of a process is its state
  - **Text segment** : the compiled code
  - **Data and bss segments:** all global variables and data structures
  - **Heap segment** : dynamically allocated data
  - **Stack segment** : local variables in the running function
  - **Machine registers,** e.g., Instruction Register
- When a process is pre-empted, the OS does a **context switch**, causing the system to execute in another process context (*there is a performance overhead*)
  - In a context switch, the OS has **to save all the information needed** so that it can switch back to the earlier process and resume
  - In a context switch, the OS has to **resume back** the new running process

Process States

- **User mode Running** : this is the normal execution of a process. A mode switch to the kernel mode is needed when a system call is performed, e.g., printf() or when the process has used its allowed time slot (interrupt) or when an interrupt occurs
- **Kernel mode Running:** it can decide whether to run, pre-empt, terminate, resume or block a process



Scheduling Algorithms

- A process scheduler schedules processes using a scheduling algorithm
  - **First Come, First Served** algorithm
  - **Shortest Job First** algorithm
  - **Priority Scheduling** algorithms
    - Each process is assigned a priority
    - Processes with high priority are executed first
  - **Shortest remaining time** algorithms
  - **Round Robin** algorithm
    - All processes are assigned to equal time slots and executed one after another
- The aforementioned algorithms can be either
  - **Pre-emptive:** A process might be pre-empted by a higher priority process
  - **Non Pre-emptive:** Once a process starts, it cannot stop, until it completes its allotted time

System Calls

- A System Call is a mechanism that provides the interface between a process and the OS.
- System calls allow user-level processes to request services from the kernel of the OS
- Programs interact with the OS via system calls
- Example of system calls are
  - fork(), exit(), open(), read(), write(), sleep(), getpid(), etc

- System Call steps
  - The user process runs at user mode until a system call is found
  - The system call is executed at kernel mode
  - Control returns to the user process

Note

- The following functions are hard to set up and this is why ***the source code will always be given to you, so don't worry***.
- The aim of this session is to understand how processes work, not to develop such code

Getting the Process' ID

- Every process on the system has a unique process ID number, aka ***pid***
  - ***pid*** is an integer
  - You can get pid via the ***getpid*** system call

```
#include <stdlib.h> //needed for exit()
#include <unistd.h> //needed for getpid()
#include <stdio.h> //needed for printf()

int main() {
printf ("my process ID is %d ",getpid() );
exit(0); //this function terminates the process immediately.}
```

Cloning a process

- You can create a process via the ***fork*** **system call** which clones a process into two processes running the same code
- ***Fork*** returns '0' to the child process and the child's '***pid***' to the parent process.
  - In a failure, -1 is returned.
- No data are shared after ***fork().***
- The process creator is called the parent and the new process is the child
- The parent defines the resources and privileges to its children
- A parent can either wait for the child process to complete or continue in parallel

Cloning a process – 1st Activity


```
#include <stdlib.h>    /* needed to define exit() */
#include <unistd.h>    /* needed for fork() and getpid() */
#include <stdio.h>     /* needed for printf() */
int main( ) {
    int pid;    /* process ID */

    pid = fork();
    printf("\nHello there\n");

    if (pid==0) { /* if you are the child process do (a fork returns 0 to the child) */
        printf("I am the child process: pid=%d\n", getpid());        }

    else if (pid==-1){    /* something went wrong */
        perror("fork"); //this will print an error message
        exit(1);}        //terminate the process (1 means not successfully)

    else{    /* if you are the parent process do (a fork returns a pid to the parent) */
        printf("I am the parent process: pid=%d, child pid=%d\n", getpid(), pid);        }
    exit(0); } //successfully terminate the process
```

Question: What does this program print?

Hello there
I am the parent process: pid11595, child pid=11596

Hello there
I am the child process: pid 11596

2nd Fork activity
What does this program do?


```
int  main(int argc, char *argv[]) {
    int pid;    /* process ID */
     int a=1, b=0;

    pid = fork();

    a++;
    printf("\nJust after fork a=%d - b=%d\n",a,b);

    if (pid==0) { //if you are the child process
      /* a fork returns 0 to the child */
        printf("I am the child process: pid=%d\n",
getpid());
        a++; b--;
        printf("\nIn child process a=%d\n",a);
        break;
}
Else if (pid==-1){    /* something went wrong
*/
        perror("fork");
        exit(1);
   }
Else { // a fork returns a pid to the parent
        printf("I am the parent process: pid=%d,
child pid=%d\n", getpid(), pid);
        a++; b++;
        printf("\nIn parent process a=%d\n",a);
        break;
    }

    printf("\nJust before end a=%d -
b=%d\n",a,b);

    exit(0);
}
```

2nd Fork activity
What does this program do? (2)

//Child Process
int a=1, b=0

a++; //a=2
Just after fork a=2 - b=0

I am the Child process : pid=…
a++; //a=3
b--; //b=-1
Child process a=3

Just before end a=3 – b=-1

//Parent Process
int a=1, b=0

a++; //a=2
Just after fork a=2 - b=0

I am the Parent process : pid=…
a++; //a=3
b++; //b=1
Child process a=3

Just before end a=3 – b=1

execve() system call

- Fork clones the original process
- Execve() system call replaces the current process with a new one.

```
//program1.c
#include <stdio.h>

int main(){
printf("\nHi from Program #1\n");
}
```

```
gcc program1.c -o program1
gcc program2.c -o program2
./program2
Hi from Program #1
```

```
//program2.c
#include <stdio.h>
#include <unistd.h> //for execve

int main(){
char *args[] = {0};
execve("program1",args,NULL);
printf("\n Error\n");
}
```

Execlp system call

- In addition to execve() system call, there are other similar routines such as execlp and execvp.
- Execlp allows us to specify all the arguments as parameters to the function and therefore it is easier to use

Program's output:

About to
Hi There

```
int main(int argc, char *argv[]) {

char *temp1,*temp2; //these are the memory addresses that the strings are stored in memory
temp1="Hi";     temp2="There";

    printf("About to \n");
    execlp("echo", "e", temp1, temp2, NULL); //the 1st argument is the command to run, the
2nd is the command's name, the others are the arguments passed to execlp. the last must be
NULL
    perror("execlp");    /* if we get here, execlp failed */
    exit(1);
}
```

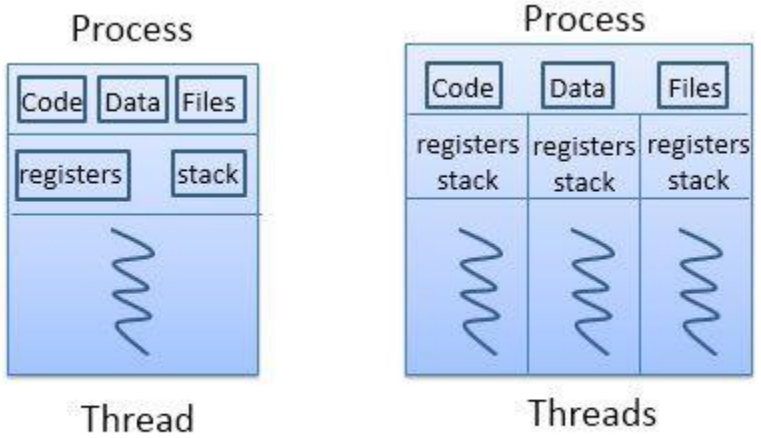
Activity – What does this program do?


```
int main() {
    int pid;    /* process ID */

    if (pid = fork()) { //if you are the child process
        funct1();
       }

    elseif (pid == -1){ //if you are the parent process
        perror("fork");
        exit(1); }

   else { // a fork returns a pid to the parent
        sleep(5);    /* sleep for 5 seconds */
        printf("Parent is still here!\n");
        break;     }
    exit(0);
}
```

```
void funct1( ) {

 execlp("echo", "echo_name", "Cheers", "from
child !", (char*)0);
    perror("execlp"); // execlp failed
    exit(1);
 }
```

What is difference between thread, process and program?

- **Program:**
  - Program is an executable file containing the set of instructions written to perform a specific job on your computer
  - For example, *skype.exe* is an executable file containing the set of instructions which help us to run *skype*
- **Process:**
  - Process is an executing instance of a program
  - For example, when you double click on the *skype.exe* on your computer, a process is started that will run the skype program
- **Thread:**
  - Thread is the smallest executable unit of a process
  - For example, when you run skype program, OS creates a process and starts the execution of the main thread of that process
  - A process can have multiple threads
  - All threads of the same process share memory of that process



Multithreading

Further Reading

- Chapter 3 and chapter 4 in Operating Systems, Internals and Design Principles, available at https://dinus.ac.id/repository/docs/ajar/Operating_System.pdf
- POSIX Threads Programming, available at https://computing.llnl.gov/tutorials/pthreads/
- POSIX thread (pthread) libraries available at https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

Thank you