Attendance Barcode

COMP2007: Operating Systems & Concurrency
Week 3 – 3:00pm Monday – 09 October 2023

valid for 15 minutes from 2:55pm
generated 2023-10-08 03:10

Operating Systems and Concurrency

Processes 1

COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

Recap

Last Lecture

- **Kernel mode** code has more privileges than user code
- **Interrupts** change the flow of execution to invoke kernel code
- **System calls** allow us to run kernel code to access services of the operating system

Goals for Today

- Introduction to **processes** and their **implementation**
- Process **states** and state **transitions**
- **System calls** for process management

Processes

Definition

- **A process is an abstraction of a running instance of a program**
  - A program is **passive** and “sits” on a disk
  - A process has **control structures** associated with it, may be **active**, and may have **resources** assigned to it (e.g. I/O devices, memory, processor)
- All the information necessary to administer a process is stored by the kernel in a **process control block (PCB).**
- All the process control blocks are recorded in the **process table.**

Processes

Memory Image of Processes

- A **process' memory image** contains:
  - The program **code** (could be shared between multiple processes running the same code)
  - A **data** segment, **stack** and **heap**
- Every process has its own **logical address space**, in which the **stack** and **heap** are placed at **opposite sides** to allow them to grow

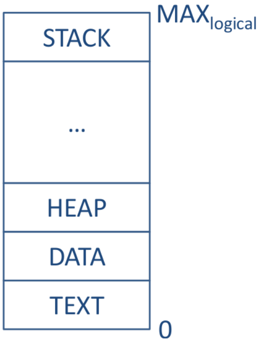
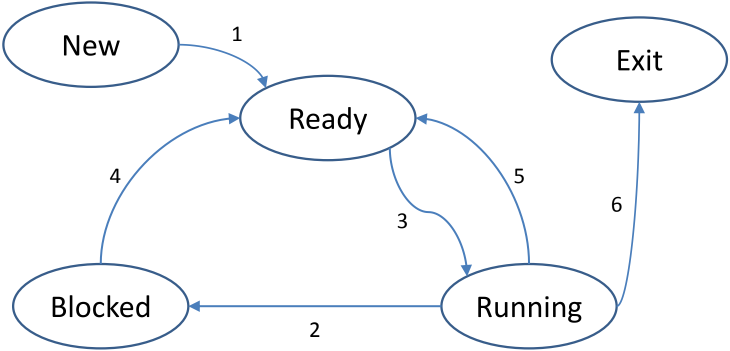


Figure: Representation of a process in memory

Process States and Transitions

Diagram



Process States and Transitions

States

- A **new** process has just been created. It has a PCB and is waiting to be admitted, although it may not yet be in memory.
- A **ready** process is waiting for CPU to become available.
- A **running** process is currently having its instructions executed by the CPU.
- A **blocked** process cannot continue, e.g. is waiting for I/O
- A **terminated** process is no longer executable. The data structures - PCB - may be temporarily preserved.

Process States and Transitions

States

- A **new** process has just been created. It has a PCB and is waiting to be admitted, although it may not yet be in memory.
- A **ready** process is waiting for CPU to become available.
- A **running** process is currently having its instructions executed by the CPU.
- A **blocked** process cannot continue, e.g. is waiting for I/O
- A **terminated** process is no longer executable. The data structures - PCB - may be temporarily preserved.
- A **suspended** process is swapped out (not discussed further)

Process States and Transitions

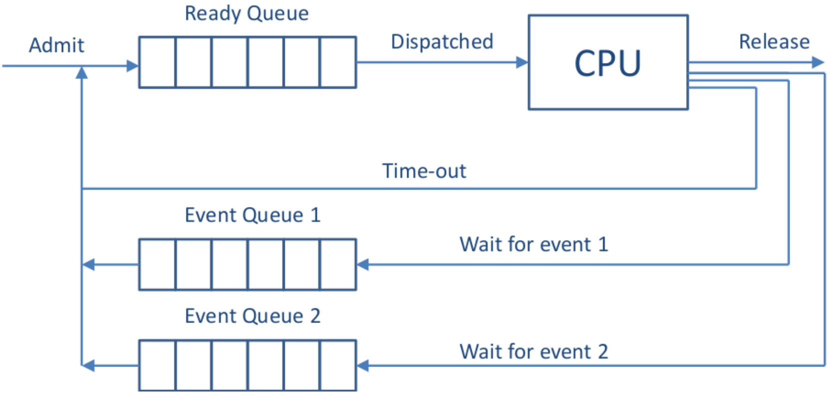
Transitions

- State transitions include:
  1. **New** $\rightarrow$ **ready**: admit the process and commit to execution
  2. **Running** $\rightarrow$ **blocked**: e.g. process is waiting for input or carried out a system call
  3. **Ready** $\rightarrow$ **running**: the process is selected by the **process scheduler**
  4. **Blocked** $\rightarrow$ **ready**: event happens, e.g. I/O operation has finished
  5. **Running** $\rightarrow$ **ready**: the process surrenders the CPU, for example due to an **interrupt** or by **pause**
  6. **Running** $\rightarrow$ **exit**: process has finished, e.g. program ended or exception encountered
- **Interrupts and system calls** drive these transitions.

Exam 2013-2014: List the 5 process states and explain the transitions between them

Process States and Transitions

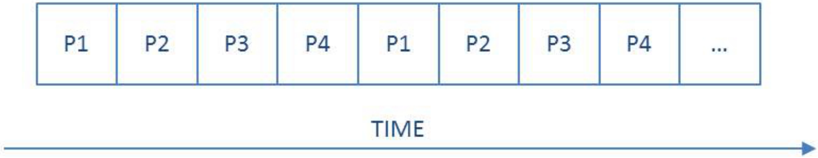
OS Queues



Context Switching

Multi-programming

- Modern computers are **multi-programming** systems
- Assuming a **single processor system**, the instructions of individual processes are executed **sequentially**
    - Multi-programming goes back to the **"MULTICS"** age
    - Multi-programming is achieved by **interleaving** the execution of processes, dividing the CPU time into **time-slices**
    - Control is exchanged between processes via a procedure known as **context switching**
    - A **trade-off** exists between the length of the **time-slice** and the **context switch time**
    - **True parallelism** requires **hardware support**



Context Switching

Multi-programming (Cont'ed)

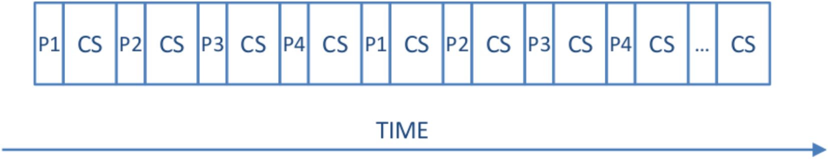
- When a **context switch** takes place, the system **saves the state** of the old process and **loads the state** of the new process (creates **overhead**)
  - **Saved** $\Rightarrow$ the process control block is **updated**
  - **(Re-)started** $\Rightarrow$ the process control block **read**

Context Switching

Multi-programming (Cont'ed)

Short time slices result in **good response times** but **low effective utilisation**. For example, assume both context switches and time slices take $1ms$. Then:

- It will take $99 \times (1 + 1) = 198ms$ for the last of 100 processes to start running.
- $\frac{1}{1+1} = 0.5$ of the CPU time is doing useful work.

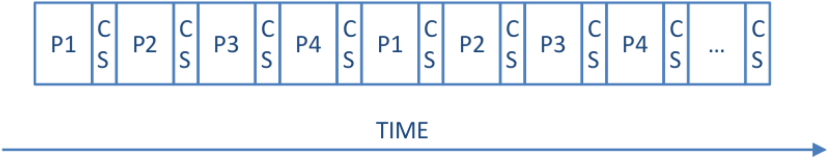


Context Switching

Multi-programming (Cont'ed)

Long time slices result in **poor response times** but **better effective utilisation**. For example, assume context switches take $1ms$ and time slices are $100ms$. Then:

- It will take $99 \times (100 + 1) = 9999ms$ for the last of 100 processes to start running.
- $\frac{100}{1+100} = 0.99$ of the CPU time is doing useful work



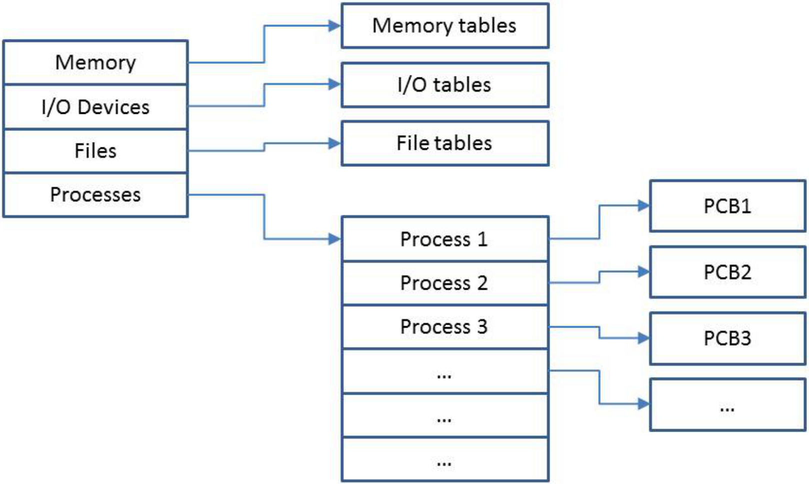
Context Switching

Multi-programming (Cont'ed)

- A **process control block** contains three types of **attributes**:
  - **Process identification** (PID, UID, Parent PID)
  - **Process control information** (process state, scheduling information, etc.)
  - **Process state information** (user registers, program counter, stack pointer, program status word, memory management information, files, etc.)
- **Process control blocks** are **kernel data structures**, i.e. they are **protected** and only accessible in **kernel mode**!
  - Allowing user applications to access them directly could **compromise their integrity**
  - The **operating system manages** them on the user's behalf through **system calls** (e.g. to set **process priority**)

Process Implementation

Tables and Control Blocks



Process Implementation

Tables and Control Blocks

- An operating system **maintains information** about the status of "resources" in **tables**
  - **Process tables** (process control blocks)
  - **Memory tables** (memory allocation, memory protection, virtual memory)
  - **I/O tables** (availability, status, transfer information)
  - **File tables** (location, status)
- The **process table** holds a **process control block** for each process, allocated upon **process creation**
- Tables are maintained by the **kernel** and are usually **cross referenced**

Context Switching

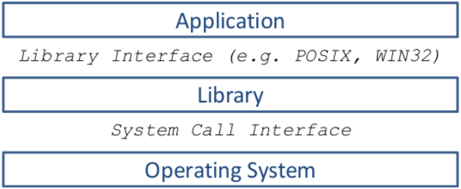
Switching Processes

```
1. Save process state (program counter, registers)
2. Update PCB (running -> ready/blocked)
3. Move PCB to appropriate queue (ready/blocked)
4. Run scheduler, select new process
5. Update to running state in the new PCB
6. Update memory management unit (MMU)
7. Restore process
```

System Calls

Process Creation

- The true system calls are **"wrapped"** in the **OS libraries** (e.g. `libc`) following a well defined interface (e.g. `POSIX`, `WIN32` API)
- For example, on Unix-like operating systems `fork` is called to create a copy of a process. On Linux, the underlying system call used to implement `fork` is `clone`.



System Calls

Process Termination

- System calls via `exit` and `abort` *can* be used to explicitly **notify the OS** that the **process has terminated**
  - Resources must be de-allocated
  - Output must be flushed
  - Process admin may have to be carried out
- A system calls to terminate other processes:
  - UNIX/Linux: `kill()`
  - Windows: `TerminateProcess()`

Process Creation in Linux

Fork

- `fork()` creates an **exact copy** of the current process
  - The first instruction carried out by the child is the first one after the `fork` call
- `fork()` returns the **process identifier** of the child process **to the parent process**.
- `fork()` **returns 0** to the **child process**.

Process Creation in Linux

The Fork and Exec Pattern

A common pattern is the following sequence:

1. Call `fork()` to create an exact copy of the current process.
2. In the child process call one of the "exec" functions to replace the current process with a new program.

Process Creation in Linux

The Fork and Exec Pattern

A common pattern is the following sequence:

1. Call `fork()` to create an exact copy of the current process.
2. In the child process call one of the “exec” functions to replace the current process with a new program. E.g. `execl("/bin/ls", "ls", "-l", 0)` replaces the current process with ls,

Process Creation in Linux

The Fork and Exec Pattern

A common pattern is the following sequence:

1. Call `fork()` to create an exact copy of the current process.
2. In the child process call one of the “exec” functions to replace the current process with a new program. E.g. `execl("/bin/ls", "ls", "-l", 0)` replaces the current process with ls,

This is a typical pattern of calls in a Unix shell such as bash.

Processes

Process Creation in Linux

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

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

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

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```


Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

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

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```


Processes

Process Creation in Linux

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

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Processes

Process Creation in Linux

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t const pid = fork();
  if(pid < 0){
    printf("Fork failure\n");
    return -1;
  } else if(pid == 0) {
    printf("Child process\n");
    execl("/bin/ls", "ls", "-l", 0);
  } else {
    int status = 0;
    waitpid(pid, &status, 0);
    printf("Child %d returned %d\n", pid, status);
  }
}
```

Test your understanding

- Why might you run `fork` without running a subsequent exec?
- Do you always need to call `exit` to end a process?
- Why does a process control block contain data about register contents?
- Why might it be useful to retain a process control block for a terminated process?