

# Processes

— DM510 Operating Systems

— Lars Rohwedder



macOS



iOS

## Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides:  
<https://www.os-book.com/OS10/slide-dir/index.html>

## Today's lecture

- > Wrapping up missing parts from last lecture (debugging, tracing, bootloader)
- > Introduction to first programming project
- > Chapter 3 of course book

## Basics

## Processes and programs

- > **Program:** executable file, passive, typically in secondary storage
- > **Process:** active instance of a program
- > There may be multiple processes corresponding to the same program
- > A process can have multiple **threads** of execution ~→ next lecture

# Processes and programs

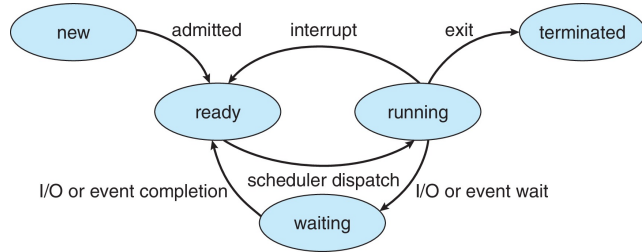
- > **Program:** executable file, passive, typically in secondary storage
- > **Process:** active instance of a program
- > There may be multiple processes corresponding to the same program
- > A process can have multiple **threads** of execution ~→ next lecture

## Today

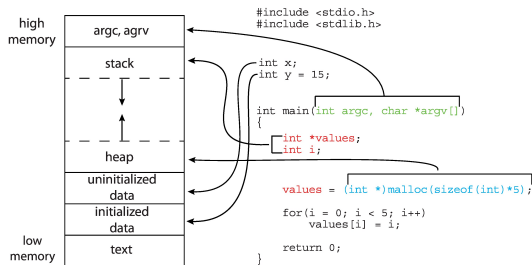
Only single-threaded processes

# States of a process

- > **New:** Loaded, but not executing, yet
- > **Running:** currently executing on CPU
- > **Waiting:** requires event to occurs to continue executing
- > **Ready:** can execute, but currently is not
- > **Terminated:** process has ended



# Data of a process



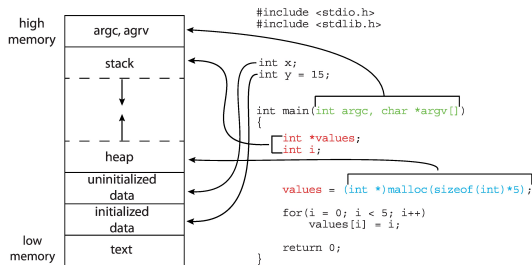
Memory layout of C program

## Context

- > Current value of CPU **registers** including program counter
- > The program code/instructions, called **text section**
- > **Stack**: function parameters, local variables, return addresses
- > **Data section**: global variables
- > **Heap**: dynamically allocated memory



# Data of a process



Memory layout of C program

## Context

- > Current value of CPU **registers** including program counter
- > The program code/instructions, called **text section**
- > **Stack**: function parameters, local variables, return addresses
- > **Data section**: global variables
- > **Heap**: dynamically allocated memory

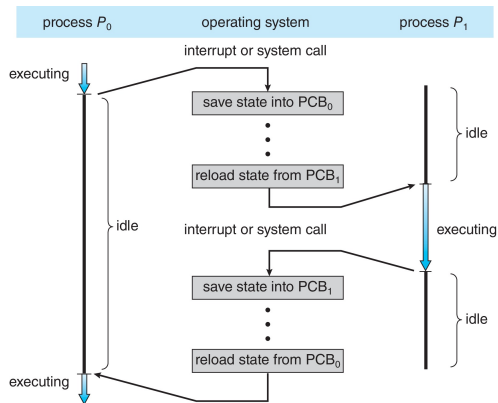
## Process control block (PCB)

Data about each process stored by kernel in main memory

- > **Process status**
- > **Scheduling information**: priority, etc.
- > **Accounting information**: elapsed time, CPU time used, etc.
- > **I/O status information**: devices allocated to process, open files, etc.
- > **Saved register values and other context data** when process is idle

# Context switch

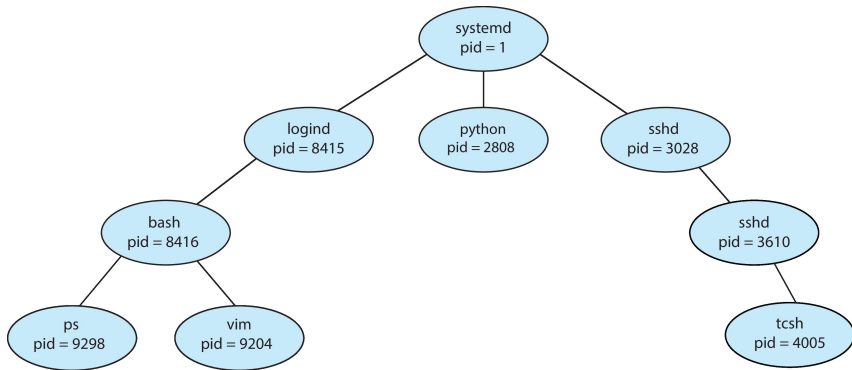
- > When kernel preempts a process, it must save its context to be able to restore it later
- > Many **context switches** can impact a system's overall performance
- > Some architectures have hardware support like multiple register sets



## Process Creation and Termination

## Process tree

- > A process can create other processes through system calls
- > This leads to a parent-child relationship among processes



## Fork system call

- > In Unix systems, the **fork()** system call is used to create a new process
- > It creates an exact copy of the caller, including context (data section, heap, stack, etc.)
- > Only difference is value returned by **fork()** will be different: zero for child process, process id of child for the parent process
- > Often system call to the loader **exec()** follows immediately after forking

### fork in C

```
int main() {
    pid_t pid;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "error");
    }
    else if (pid == 0) {
        printf("child-process");
    }
    else {
        printf("parent-process");
        wait(NULL)
        printf("child-terminated");
    }
    return 0;
}
```

## Termination

- > A process terminates by using the `exit()` system call (sometimes implicit when returning from `main()` function). It provides a return value that typically indicates success or failure

## Termination

- > A process terminates by using the `exit()` system call (sometimes implicit when returning from `main()` function). It provides a return value that typically indicates success or failure
- > Return code is kept by kernel until collected by parent with `wait()` system call

## Termination

- > A process terminates by using the `exit()` system call (sometimes implicit when returning from `main()` function). It provides a return value that typically indicates success or failure
- > Return code is kept by kernel until collected by parent with `wait()` system call
- > Process that has ended, but return code has not been collected is called **zombie**



## Termination

- > A process terminates by using the `exit()` system call (sometimes implicit when returning from `main()` function). It provides a return value that typically indicates success or failure
- > Return code is kept by kernel until collected by parent with `wait()` system call
- > Process that has ended, but return code has not been collected is called **zombie**
- > Process whose parent terminates without calling `wait()` is an **orphan**

## Termination

- > A process terminates by using the `exit()` system call (sometimes implicit when returning from `main()` function). It provides a return value that typically indicates success or failure
- > Return code is kept by kernel until collected by parent with `wait()` system call
- > Process that has ended, but return code has not been collected is called **zombie**
- > Process whose parent terminates without calling `wait()` is an **orphan**
- > Parent can terminate child using `abort()` system call

# Termination

- > A process terminates by using the `exit()` system call (sometimes implicit when returning from `main()` function). It provides a return value that typically indicates success or failure
- > Return code is kept by kernel until collected by parent with `wait()` system call
- > Process that has ended, but return code has not been collected is called **zombie**
- > Process whose parent terminates without calling `wait()` is an **orphan**
- > Parent can terminate child using `abort()` system call

## Reasons for aborting a process

- > Task performed by process is no longer needed
- > Kernel needs to reclaim resources
- > Sometimes child process is not allowed to continue when parent terminates

## Cooperation and Communication

# Cooperation

- > Some processes work independently, other cooperate regarding their tasks
- > Cooperation requires means of **communication** provided by the kernel

# Cooperation

- > Some processes work independently, other cooperate regarding their tasks
- > Cooperation requires means of **communication** provided by the kernel

## Example: Chrome Browser

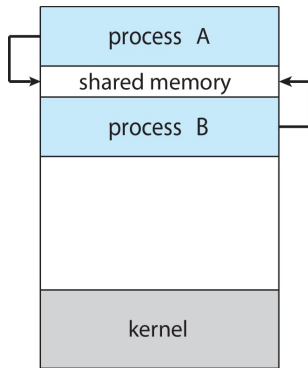


An extreme example of cooperation is Google's Chrome web browser

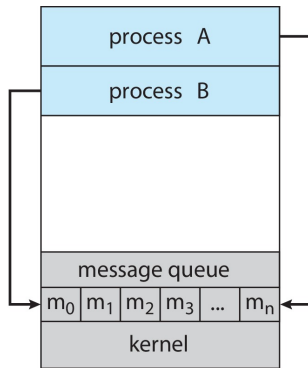
- > Separate renderer process for each tab + main process + plug-in processes
- > **More secure** by restricting privileges of websites
- > **More reliable** since single malfunctioning websites does not crash entire browser

## Communication models

Two main variants of communication: (a) **shared memory** and (b) **message passing**



(a)



(b)

## Producer-consumer with shared memory

The following example has one process produce items and the other consume them. Both processes have access to the following data:

```
#define BUFLLEN 10
item buffer[BUFLLEN];
int in = 0;
int out = 0;
```

### Producer

```
while (true) {
    item next_produced = produce();
    while (((in + 1) % BUFLLEN) == out)
        ; /* busy waiting */
    buffer[in] = next_produced;
    in = (in + 1) % BUFLLEN;
}
```

### Consumer

```
while (true) {
    while (in == out)
        ; /* busy waiting */
    item next_consumed = buffer[out];
    out = (out + 1) % BUFLLEN;
    consume(next_consumed);
}
```



## Race condition in shared memory

Without appropriate measures (see synchronization lecture), preemption can lead to errors

Consider two producers running the previous code simultaneously:

```
> in = 0  
> out = 0  
> buffer[0] = uninitialized  
> buffer[1] = uninitialized
```

### Producer A

```
>  
>  
>  
>
```

### Producer B

```
>  
>  
>  
>
```

## Race condition in shared memory

Without appropriate measures (see synchronization lecture), preemption can lead to errors

Consider two producers running the previous code simultaneously:

```
> in = 0  
> out = 0  
> buffer[0] = itemA  
> buffer[1] = uninitialized
```

### Producer A

```
> buffer[in] = itemA;  
>  
>  
>
```

### Producer B

```
>  
>  
>  
>
```

## Race condition in shared memory

Without appropriate measures (see synchronization lecture), preemption can lead to errors

Consider two producers running the previous code simultaneously:

```
> in = 0  
> out = 0  
> buffer[0] = itemB  
> buffer[1] = uninitialized
```

### Producer A

```
> buffer[in] = itemA;  
>  
>  
>
```

### Producer B

```
>  
> buffer[in] = itemB;  
>  
>
```

## Race condition in shared memory

Without appropriate measures (see synchronization lecture), preemption can lead to errors

Consider two producers running the previous code simultaneously:

```
> in = 1  
> out = 0  
> buffer[0] = itemB  
> buffer[1] = uninitialized
```

### Producer A

```
> buffer[in] = itemA;  
>  
>  
>
```

### Producer B

```
>  
> buffer[in] = itemB;  
> in = (in + 1) % BUFLLEN;  
>
```

## Race condition in shared memory

Without appropriate measures (see synchronization lecture), preemption can lead to errors

Consider two producers running the previous code simultaneously:

```
> in = 2  
> out = 0  
> buffer[0] = itemB  
> buffer[1] = uninitialized
```

### Producer A

```
> buffer[in] = itemA;  
>  
>  
> in = (in + 1) % BUFLLEN;
```

### Producer B

```
>  
> buffer[in] = itemB;  
> in = (in + 1) % BUFLLEN;  
>
```

## Race condition in shared memory

Without appropriate measures (see synchronization lecture), preemption can lead to errors

Consider two producers running the previous code simultaneously:

```
> in = 2  
> out = 0  
> buffer[0] = itemB  
> buffer[1] = uninitialized
```

### Producer A

```
> buffer[in] = itemA;  
>  
>  
> in = (in + 1) % BUFLLEN;
```

### Producer B

```
>  
> buffer[in] = itemB;  
> in = (in + 1) % BUFLLEN;  
>
```

Only some instructions are guaranteed to be executed **atomically** (not preempted). Even within a single line the code can be preempted

## Message passing

kernel provides system calls `send(link, message)` and `receive(link, &message)`

### Variants

- > fixed length or variable length messages, byte stream
- > unidirectional or bidirectional
- > means of establishing link:
  - > By process id (direct communication)
  - > Parent process creates link, which child can access (see for example ordinary pipes)
  - > Via ports or file system (see for example named pipes)
- > synchronous (wait until ) or asynchronous (continue immediately)
- > buffering: zero capacity, bounded capacity, unbounded capacity

# Message passing

kernel provides system calls `send(link, message)` and `receive(link, &message)`

## Variants

- > fixed length or variable length messages, byte stream
- > unidirectional or bidirectional
- > means of establishing link:
  - > By process id (direct communication)
  - > Parent process creates link, which child can access (see for example ordinary pipes)
  - > Via ports or file system (see for example named pipes)
- > synchronous (wait until ) or asynchronous (continue immediately)
- > buffering: zero capacity, bounded capacity, unbounded capacity

Producer-consumer with message passing:

## Producer

```
while (true) {  
    item next_produced = produce();  
    send(link , next_produced);  
}
```

## Consumer

```
while (true) {  
    item next_consumed;  
    receive(link , &next_consumed);  
    consume(next_consumed);  
}
```



## Other forms of communication

- > TCP/IP connection using sockets
- > Remote procedure calls and local procedure calls

We defer discussion to the networks lecture