50004 - Operating Systems - Lecture 2

Oliver Killane

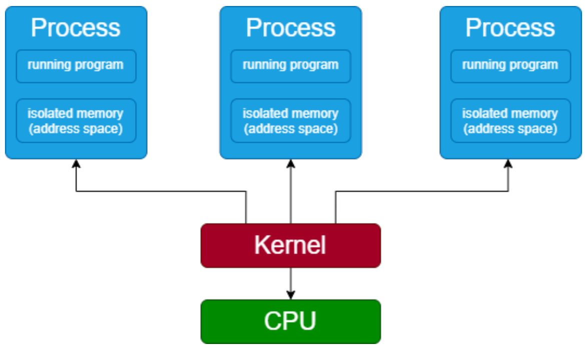
15/10/21

Lecture Recording

Lecture recording is available here

Processes

Processes are one of the oldest abstractions in computing, holding an instance of a running program.


Process
running program
isolated memory (address space)
Process
running program
isolated memory (address space)
Process
running program
isolated memory (address space)
Kernel
CPU

The process abstraction can allow and OS to run programs simultaneously, even on only one CPU core.

Each process runs on a isolated **virtual CPU**, to achieve this the CPU resources of the system are multiplexed & managed by the OS.

Why Have Processes?

- **Provide Concurrency** To ensure real concurrency works properly (programs running on multiple CPU cores).

  And to manage a single CPU core so that multiple processes can appear to run simultaneously, each process running independently.

- **Provide Isolation** Each process has its own address space, and does not need to consider other unrelated processes in order to run correctly.

- **Simplify Programming** Applications can be easily developed to run without needing to consider how resources will be managed, or what other applications may be running. Programs can be written as if they will be run without any other programs running.

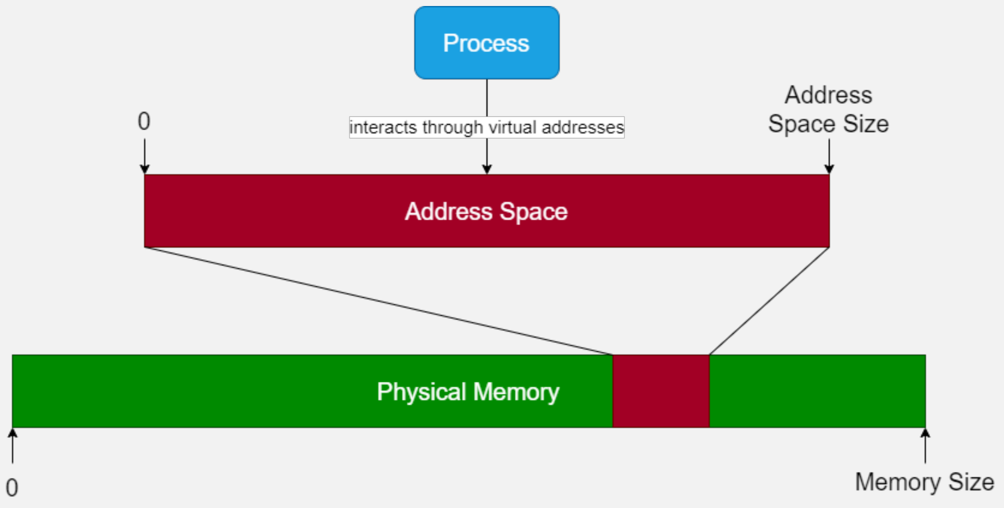
- **Better Resource Utilisation** Different processes require different resources. Hence we can more effectively increase utilisation & performance with multiple processes even on a single CPU core.

  For example a program may need to wait on I/O, rather than leaving this process idling while waiting, other programs can be scheduled to make use of resources during thwe wait.

Address Space

A process' address space is a contiguous region of memory allocated to a process by the operating system for use in storing data.

A process accesses this region using **virtual addresses** indexed from 0.



Multitasking

Time Slicing

Processes are given a set **time slice** to run on the CPU, before a **context-switch** is done to another process.

By switching between processes very quickly, the illusion of concurrency can be achieved on a single CPU core.

Context Switch

A context switch is where the CPU switches from the running one process/thread, to running another.

To achieve this the switch must:

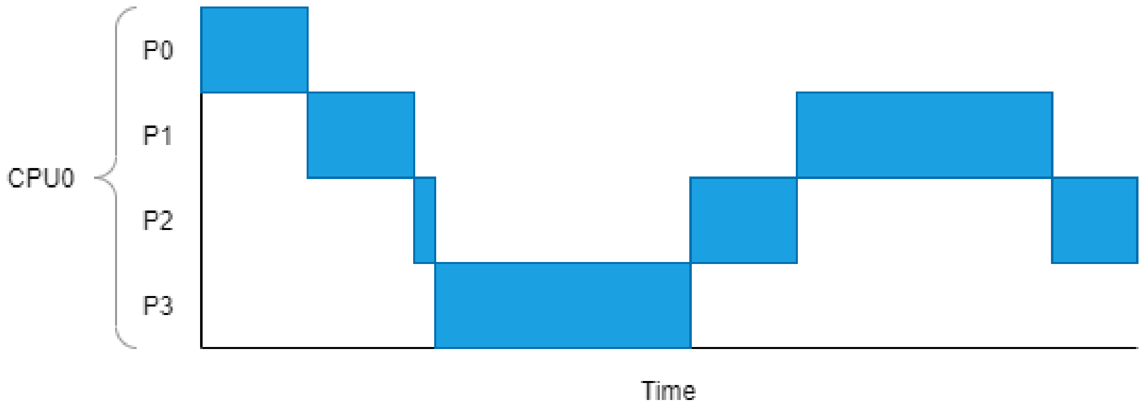
1. store the current state (Registers such as the stack pointer) such that it can be resumed later.
2. Load the previously stored state of another thread/process (registers such as stack pointer)
3. Resume processing the current thread (now switched)

This can occur due to an interrupt (e.g timer interrupts for a time slice strategy, accessing disk and other I/O etc) or in some systems when moving between **user mode** and **kernel mode** tasks.

Context switches are an essential feature of any multitasking operating system. Too many context switches can have a negative impact on performance.

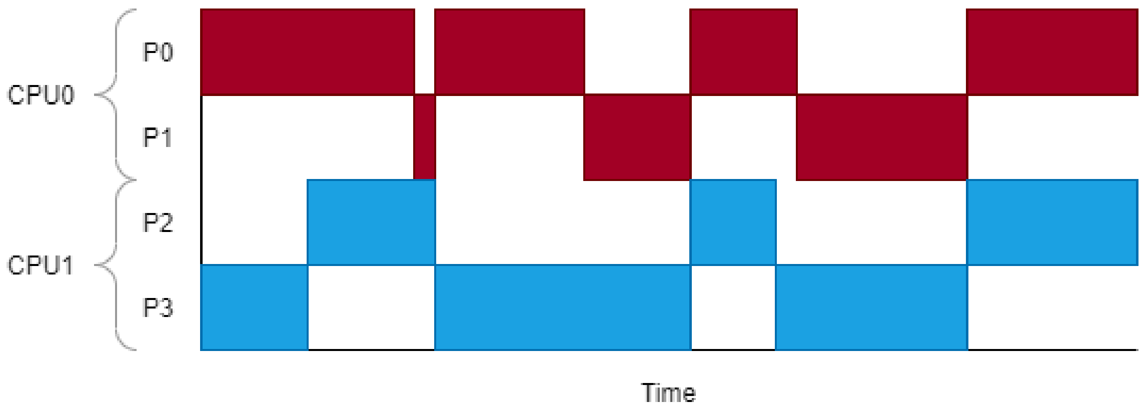
Pesudo Concurrency

Single processor, processes are interleaved to give the illusion of concurrency.



Real Concurrency

Several processors, running threads in parallel. Pesudo concurrency can also be employed (e.g in the diagram, 2 CPUs, 4 processes).



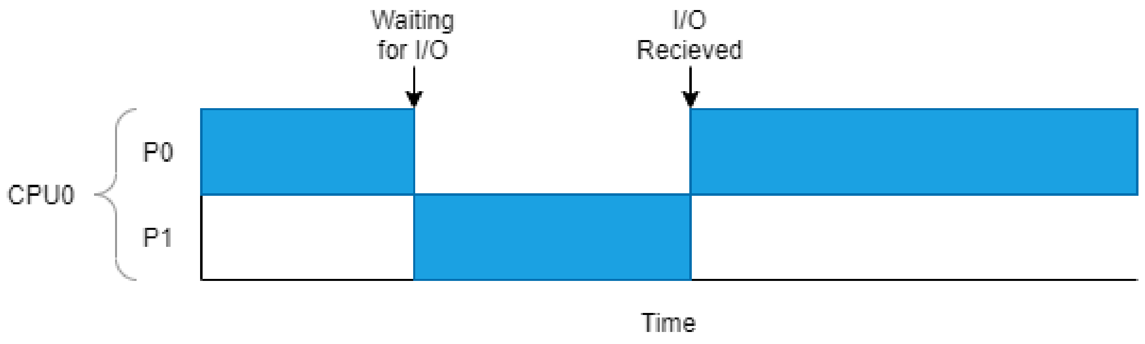
Fairness

When implementing a scheduler to determine when threads run we must consider how much time is allocated to each.

When one application is using too much CPU time, other processes may lagg (latency) and perform badly. Likewise if a process is not allocated sufficient resources.

Higher CPU Utilisation

We can switch to other threads when the current thread is waiting, instead of idling. This is beneficial so long as the context switch overhead is smaller than the utilisation that would've been lost to waiting.



We can compute an estimate for CPU utilisation on a uniprocessor:

$$\begin{aligned} n &= \text{total number of processes} \\ p &= \text{Fraction of time a process is waiting for I/O} \end{aligned}$$

$$Prob(\text{all waiting for I/O}) = p^n$$

$$CPUutilisation = 1 - p^n$$

Process Control Block

A **Process Control Block** contains a process's information:

- **Process Identification** The process's unique ID, its group, parent etc. Used to identify what resources (e.g perhiperals) the process is using.
- **Process State Data** Values of saved registers (when suspended), such as the stack pointer, page table register etc.
- **Process Control Information**
  - Scheduling state (e.g ready, suspended, blocked)
  - Priority values (for priority scheduling)
  - Related process IDs (e.g parent)
  - **IPC** information (flags, signals and messages associated with communication among independent processes)
  - Process privileges (e.g file access, peripheral access)
  - Usage Statistics such as recent CPU time, used for scheduler priority calculations.
- **File Management Info** Root directory, working directory and open file descriptors.

Context Switch Expense

Context switches are expensive, and have considerable overhead, so we must avoid them if possible.

Direct Cost: save/restore process state.
Indirect Cost: pertubation of memory caches, TLB

Transaltion Lookaside Buffers

A cache storing recent translations of virtual addresses to physical addresses (used as user programs use virtual addresses to access their own address space).

It is a hardware feature that is part of a memory-management unit (MMU) and can reside between the CPU & CPU cahce, or between CPU cahce and the main memory.

During a context switch these are typically flushed.

Process Lifecycle

Process Creation

- System initialisation (start services, e.g file system)
- User request (run a program)
- Process Request (process makes a system call to start a new process)

Daemon

A background process, typically does not interact directly with users, e.g sshd (linux) listens for connections from clients, and forks a new daemon to manage each new connection.

Destruction

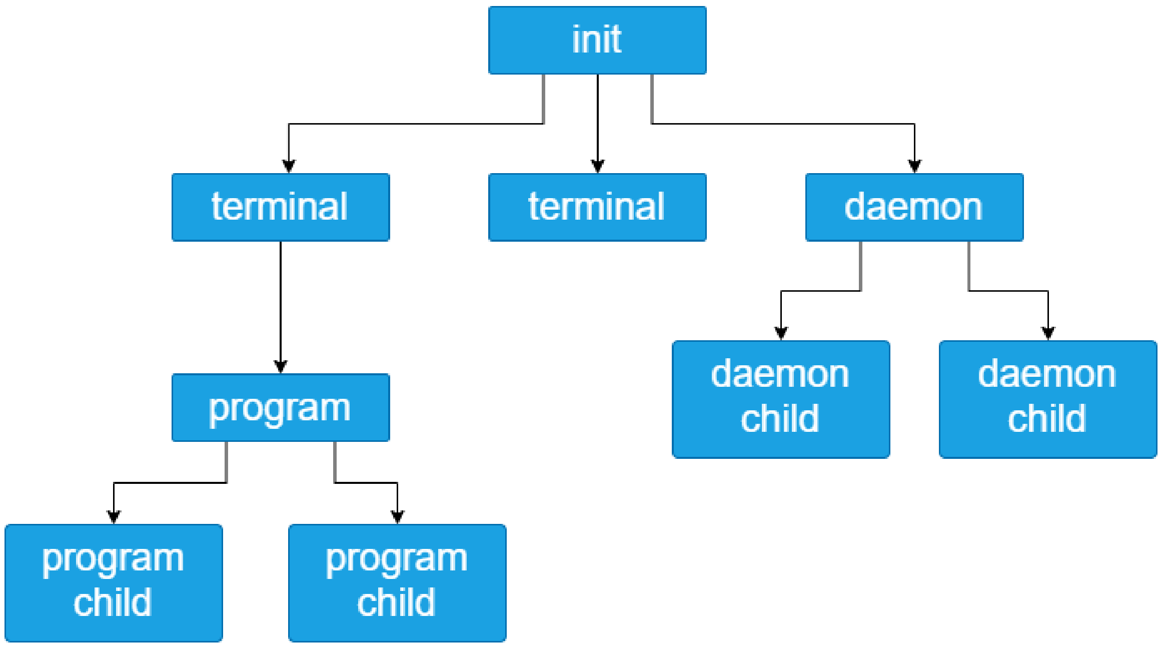
- **Completion**
  Once at the end of execution body (main function returns), the process ends.
- **System Call**

UNIX:	exit()
Windows:	ExitProcess()

- **Abnormal Exit**
  The process runs into an error or unhandled exception, and is forced to stop.
- **Aborted**
  Process stops because another process forces its execution to end. e.g parent process (terminal) forces child (program) to end.
- **Never**
  Some never terminate, many daemons need to be running at all timnes to ensure the OS or some crtitcal application being run functions as it should.

Process Hierarchies

UNIX has a strict tree of processes.



- On boot, **init** is created, this is the root of the process hierarchy tree (**process group**).

- Reads a file to determine how many processes need to be run, forks off one terminal per terminal.
- Each terminal waits for a user to login.
- On successful login, the login process executes a shell to accept commands which may in turn spawn more processes.

Windows has no such enforced hierarchy, upon spawning a process the parent is given a token/handle to control the child. This token can be transferred to other processes.

UNIX Processes

fork

```c
int fork(void)
```

Fork creates an identical copy of the process, the child process iherits resources of the parent process and is run concurrently.

Return	**Reason**
0	You are the child
-1	fork failed (out of memory, max process limit reached, etc)
n	You are the parent, pid n is the child.

```c
/* unistd provides access to the POSIX API */
#include <unistd.h>

/* fork is part of unistd */
int fork (void);

int main() {
    int child_pid = fork();

    if (child_pid != 0) {
        /* parent code here */
    } else {
        /* child code here */
    }
}
```

Executing Processes

```c
int execve(const char *path, char *const argv[], char *const envp[])
```

Changes the process image and runs new process. Arguments:

- **path** Full pathname to program to run
- **argv** arguments to pass to main
- **envp** environment variables (e.g **$PATH) $HOME**

There are many useful wrappers to make calls easier/cleaner such as; execl, execle, execvp, execv...

Waiting for Process Termination

```c
int waitpid(int pid, int* stat, int options)
```

Suspends the current process until the process **PID** has terminated.

PID	**Result**
-1	Wait for any child
0	Wait for any child in the same **process group** as the caller
$-gid$	Wait for any child with **process group** *gid*
pid	wait for process *pid*

Process Group

A collections of processes, typically used for signal distribution.

When a **process group** is signalled, every contained process receives the signal, and can further direct it to other process groups.

For example in the terminal, keyboard input can be sent to the terminal application, and the propagated to all running programs of that terminal instance.

Return value is as follows:

Value	**Result**
0	if **WNOHANG** is in set options (call should not block)
-1	if an error occured, errorno set to indicate error
pid	**PID** of process that terminated.

```c
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char** argv) {

    char* command;
    char** parameters;

    for (;;) {
        get_command(&command, &parameters);

        if (fork()) {
            waitpid(-1, &status, 0);
        } else {
            execve(command, parameters, NULL);
        }
    }

}
```

Process Termination

```c
void exit(int status)
```

Terminates the process (called implicitly when execution finishes), returns exit status to the parent process.

```
void kill(int pid, int sig)
```

Kill process using **PID**, by sending a signal to the process.

UNIX Philosophy

UNIX places value on building basic blocks to be combined into more complex programs, fork is an example of this, while execve could be used to replicate fork, it would be far more complex.

Windows does not follow this design philosophy, createProcess is the equivalent if fork and execve. It has 10 parameters for:

- program to execute
- parameters
- security attributes
- meta data for files
- priority
- pointer to info regarding new process
- and more!

```
BOOL WINAPI CreateProcess(
     __in_opt LPCTSTR lpApplicationName,
     __inout_opt LPTSTR lpCommandLine,
     __in_opt LPSECURITY_ATTRIBUTES lpProcessAttributes,
     __in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
     __in BOOL bInheritHandles,
     __in DWORD dwCreationFlags,
     __in_opt LPVOID lpEnvironment,
     __in_opt LPCTSTR lpCurrentDirectory,
     __in LPSTARTUPINFO lpStartupInfo,
     __out LPPROCESS_INFORMATION lpProcessInformation )
int pipe(int fd[2])
```

Process Communication

UNIX Signals

A limited form of **IPC** that works similarly to handler interrupts. A process can send a signal to another process if it has permission (same user for receiver or elevated privileges), the kernel can send a signal to any thread.

Signals can be triggered by:

- **Exceptions** (e.g Zero division error (SIGFPE) or segfault (SIGSEGV))

- **Events** e.g Kernel notifies a process if it writes to a closed pipe (SIGPIPE), or input (keyboard interrupt → SIGINT)
- **Programatically** Using the kill system call.

Common Signals are as follows

Signal	**Description**
SIGINT	Interrupt from keyboard
SIGABRT	Abort signal from abort
SIGFPE	Floating point exception
SIGKILL	Kill signal
SIGSEGV	Invalid memory reference
SIGPIPE	Broken pipe: write to pipe with no readers
SIGALRM	Timer signal from alarm
SIGTERM	Termination signal

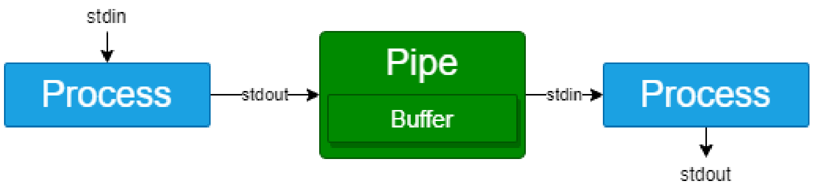
Signal Handlers

The default action for most signals is to terminate the process, however processes can register their own signal handlers.

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

void int_handler(int sig) {
    printf("SIGINT occured, but I'm ignoring it!");
}

int main(int argc, char** argv) {

    /* register handler */
    signal(SIGINT, int_handler);
    for(;;);
}
```

UNIX Pipes

A pipe connects the standard output of one process to the standard input of another.



```c
int pipe(int fd[2])
```

fd[0]	read end (closed by sender)
fd[1]	write end (closed by receiver)

When the receiver reads from an empty pipe, it is blocked until more data is written.
When the sender writes to a full pipe, it is blocked until data is read and removed from the pipe.

```c
#include<unistd.h>
#include <stdlib.h>
#include<stdio.h>

int main(int argc, char **argv) {

    /* pipe and buffer declaration */
    int fd[2];
    char buff;

    /* assert we have one argument string */
    assert(argc == 2);

    /* initialise pipe, if an error, return */
    if (pipe(fd) == -1)
        exit(EXIT_FAILURE);

    /* fork into parent and child */
    if (fork() != 0) {

        /* Parent: Sender */

        /* Close reading end (sender does not use this) */
        close(fd[0]);

        /* write the argument string to the pipe*/
        write(fd[1], argv[1], strlen(argv[1]));

        /* finished writing, now close write end */
        close(fd[1]);

        /* wait for the child to die*/
        waitpid(-1, NULL, 0);

        /* celebratory message */
        printf("Child Dead & Pipe Closed!\n");
    } else {

        /* Child: Receiver */

        /* close writing end (receiver does not use this) */
        close(fd[1]);

        /* read from pipe one character at a time, printing each char */
        while (read(fd[0], &buff, 1) > 0)
            printf("%c", buff);

        printf("\n");

        /* finished reading, now close the reading end */
        close(fd[0]);

        /* child reaches end of execution, terminates */
    }
}
```

UNIX Named Pipes

Persistent pipes outlive the processes which create them and are stored in the file system.

```bash
# create the pipe
mkfifo /some/pipe

# write to the pipe
echo "This is some text" >/some/pipe

# get text from the pipe
cat /some/pipe
```