Sistemas Operativos: The Process Abstraction

Pedro F. Souto (pfs@fe.up.pt)

February 15, 2017

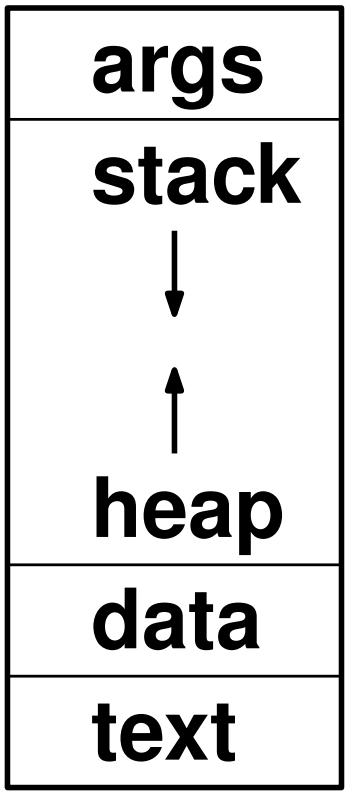
Roadmap: The Process Abstraction

(Sequential) Process

Abstracts a running program

- Processor/core state (i.e. the state of the processor registers)
- Address space
- Open files (also I/O devices)

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



Command line arguments

Activation records of called functions

Dynamically allocated data

Data allocated statically by the compiler

Program instructions

Stack

- Memory region *"that can be accessed"* on one of its ends, using the `push` and `pop` operations (Fig. 1).

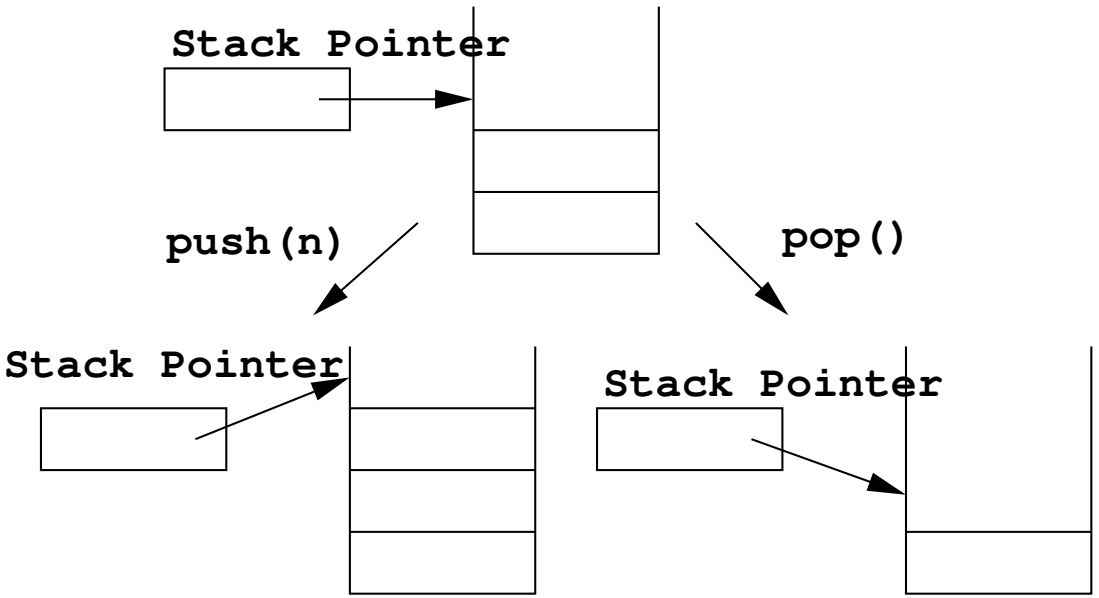


Fig. 1

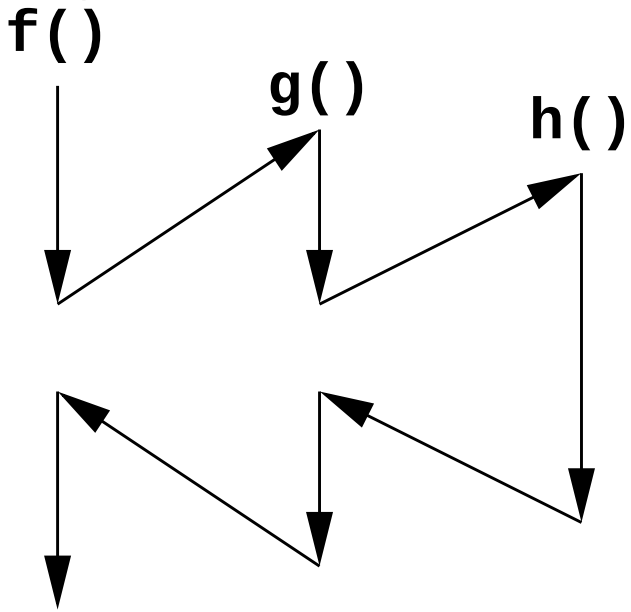


Fig. 2

- This type of access is particularly useful to implement function calls (Fig. 2).

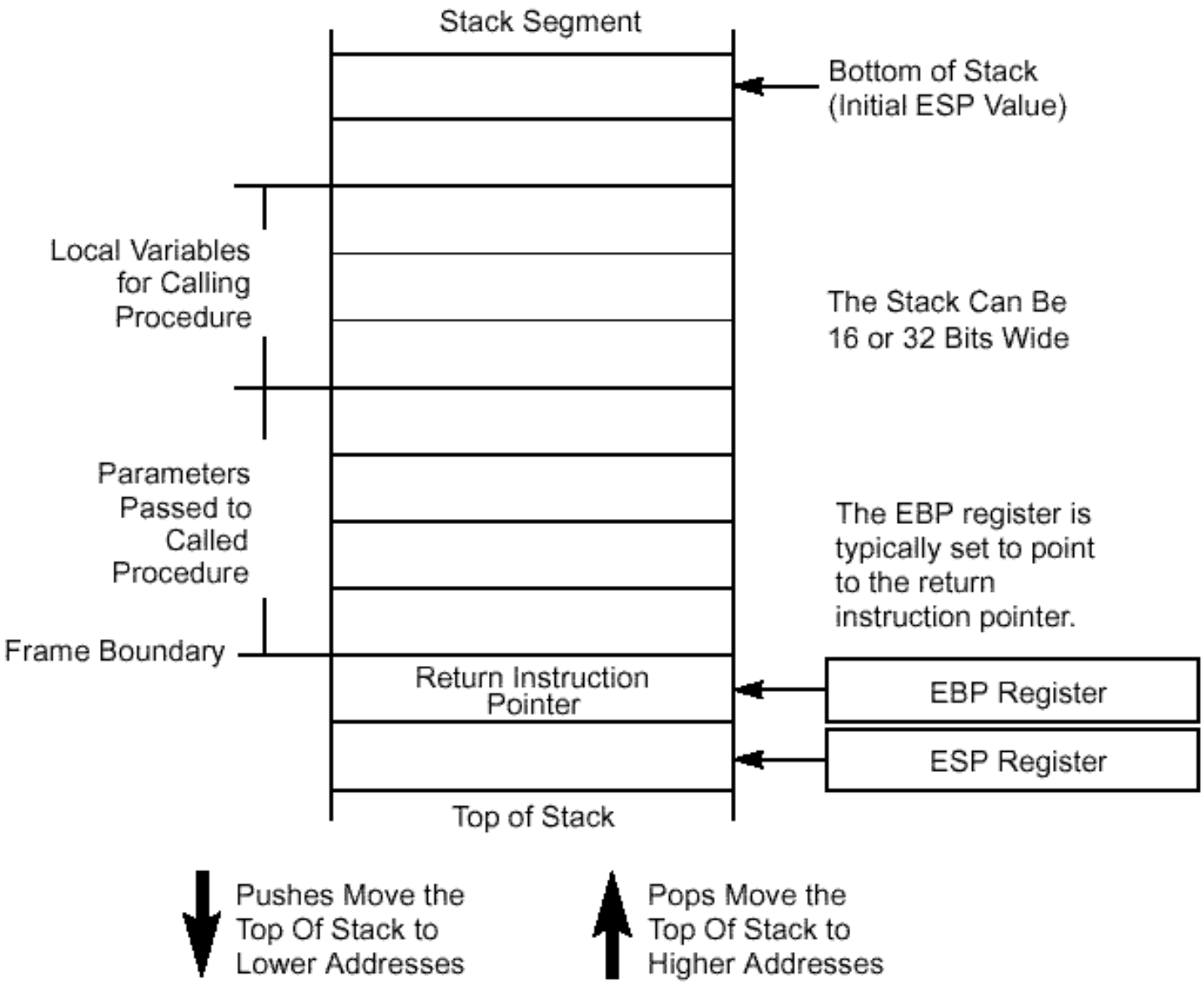
The *stack* and C functions

- In C, the *stack* is used for:
  - passing arguments;
  - saving the return address;
  - storing local variables;
  - passing the return value

  of a function. It is also used for temporarily:
  - saving the processor registers

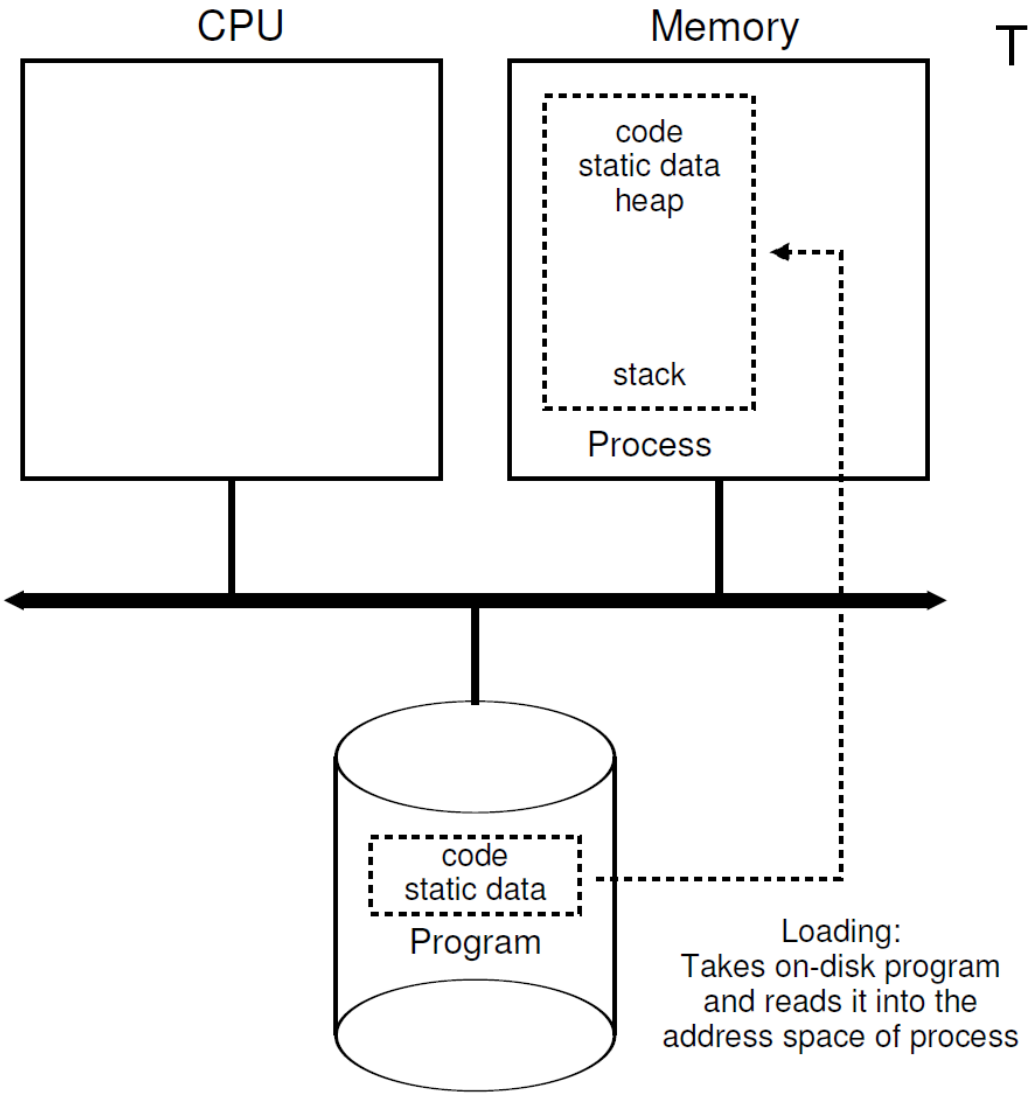
  used by a function.
- Each compiler defines a covention on how this information is stored on the stack.
  - The corresponding structure is known as the *stack frame* or the *activation record*.

i386 Stack Frame



Process Creation

- A process is different from a program



The OS:

1. loads the code and the static data into main memory;
2. initializes the **stack**, filling in the arguments for `main()`
3. may allocate some space for the **heap**
4. initializes the file descriptors for the stdin, stdout and stderr
5. sets everything up so that when the process is set to the running state it will start executing `main()`

src: Arpaci-Dusseau & Arpaci-Dusseau

Roadmap: The Process Abstraction

Unix/Linux are multiprocess OSs

```
$ ps ax | more
  PID TTY      STAT   TIME COMMAND
    1 ?        Ss     0:04 /sbin/init
    2 ?        S      0:00 [kthreadd]
    3 ?        S      0:09 [ksoftirqd/0]
    6 ?        S      0:00 [migration/0]
   11 ?        S<     0:00 [cpuset]
   12 ?        S<     0:00 [khelper]
   13 ?        S<     0:00 [netns]
   15 ?        S      0:00 [sync_supers]
   16 ?        S      0:00 [bdi-default]
   17 ?        S<     0:00 [kintegrityd]
   18 ?        S<     0:00 [kblockd]
   19 ?        S<     0:00 [kacpid]
   20 ?        S<     0:00 [kacpi_notify]
   21 ?        S<     0:00 [kacpi_hotplug]
   22 ?        S<     0:00 [ata_sff]
--More-- (238 in all)
```

- OSs support multiprogramming for reasons of **efficiency** and **convenience**.

Multiprocess and Efficiency

Problem I/O devices such as the keyboard, the mouse, or even the disk and the network interface, are much slower than the processor (and the memory)

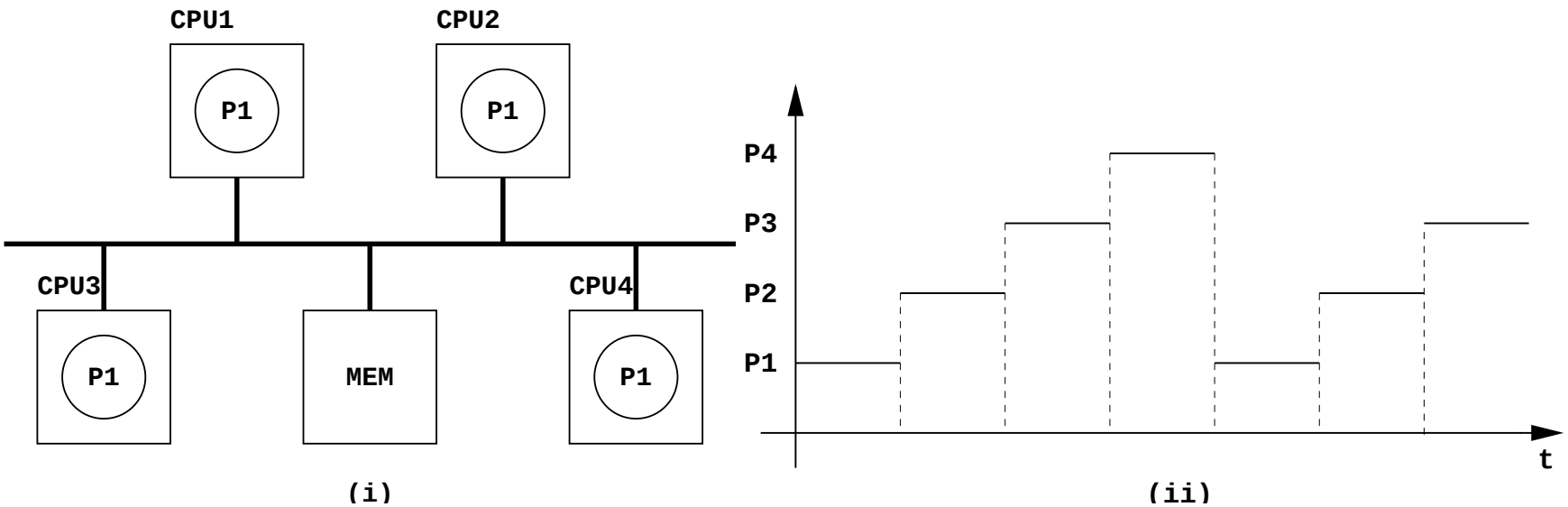
Parameter	Time
CPU cycle	1 ns (1 GHz)
Cache access	~ <10ns
Memory access	~10-100 ns
HDD access	~10 ms

Solution when a process starts an I/O operation and cannot make progress until it terminates, the OS chooses another process to run:

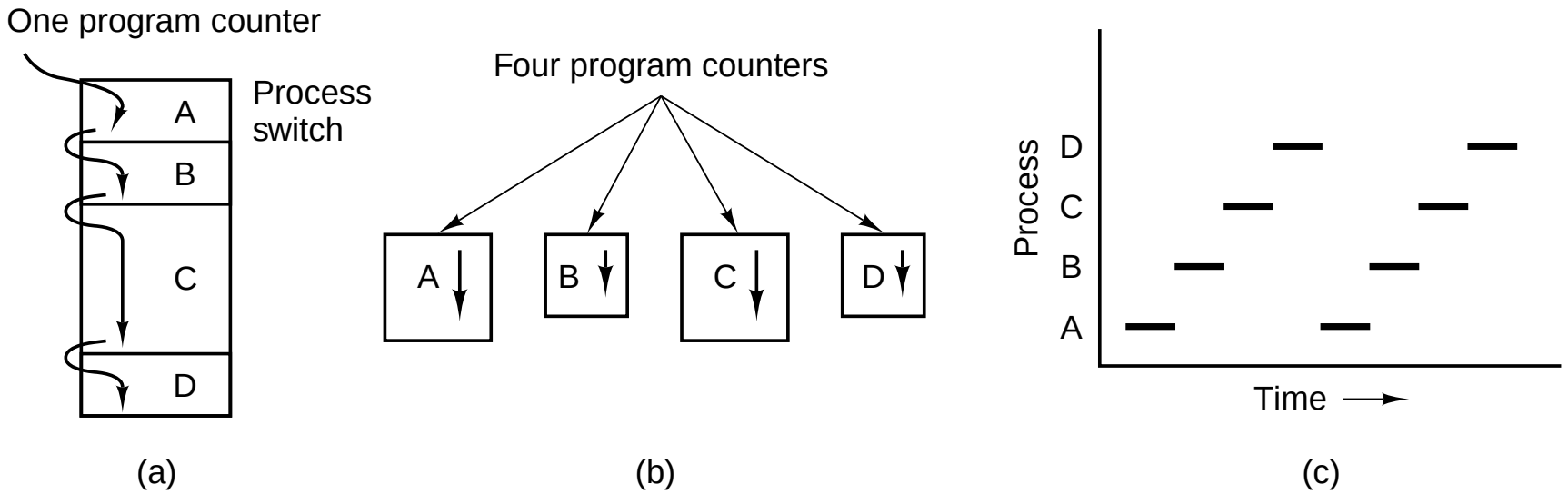
- Upon termination of the operation the device generates an interrupt

Multiprocess Execution (1/2)

- In multiprocessor systems (i), or *multicore*, several processes may run simultaneously, one in each processor/core: **real parallelism**
- In a uniprocessor system (ii), the OS assigns the processor to the different processes (the processor is a resource shared among the different processes): **pseudo-parallelism**



Multiprocess Execution (2/2)

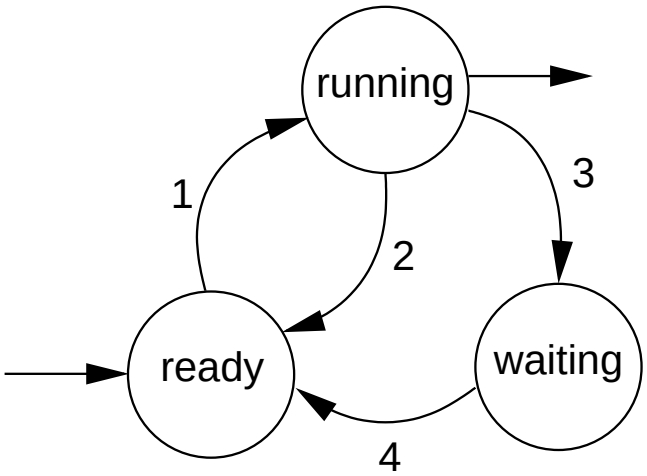


- The processor is shared among 4 processes
- The OS provides the illusion that each process executes in its own processor, i.e. each process executes in a virtual processor.

Process States

- During its lifetime a process may be in 1 of 3 states:

  running: the CPU executes the process instructions;
  waiting: the process is waiting for an external event, e.g. the termination of an I/O operation, to resume its execution;
  ready: the process is waiting for the CPU, which is executing the instructions of another process



1. The OS assigns the CPU to the process;
2. The OS preempts the process to assign the CPU to another process;
3. The process waits for an event;
4. The event the process was waiting for occurs.

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process 0 now done

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process 0 now done
5	-	Running	

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process 0 now done
5	-	Running	
6	-	Running	

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process 0 now done
5	-	Running	
6	-	Running	
7	-	Running	

Process State Traces (1/2)

Without I/O: processes use only the CPU

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process 0 now done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process 1 now done

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	and is blocked

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	and is blocked
5	Blocked	Running	and Process 1 runs

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	and is blocked
5	Blocked	Running	and Process 1 runs
6	Blocked	Running	

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	and is blocked
5	Blocked	Running	and Process 1 runs
6	Blocked	Running	
7	Ready	Running	I/O done

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	and is blocked
5	Blocked	Running	and Process 1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process 1 now done

Process State Traces (2/2)

With I/O: process 0 initiates I/O, and blocks/waits until I/O is done

Time	Process 0	Process 1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process 0 initiates I/O
4	Blocked	Running	and is blocked
5	Blocked	Running	and Process 1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process 1 now done
9	Running	-	

OS decides:

1. to run process 1, when process 0 blocks: improves CPU utilization
2. not to switch back to process 0 as soon as I/O completes

Kernel Data Structures (xv6 toy-OS)

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Roadmap: The Process Abstraction

Criação de Processos em Unix/Linux

```
#include <unistd.h>
pid_t fork(void);   /* clones the calling process *
```

- O processo criado (filho):
  - executa o mesmo programa que o programa pai;
  - inicia a sua execução na instrução que segue a `fork()`.
- O processo filho herda do pai:
  - o ambiente e “privilégios de acesso a recursos”;
  - alguns recursos, incluindo ficheiros abertos.
- Contudo, o processo filho tem os seus próprios
  - identificador;
  - espaço de endereçamento:

    após a execução de `fork()`, alterações à memória pelo *pai* não são visíveis ao *filho* e vice-versa.

Criação de Processos em Unix/Linux (cont.)

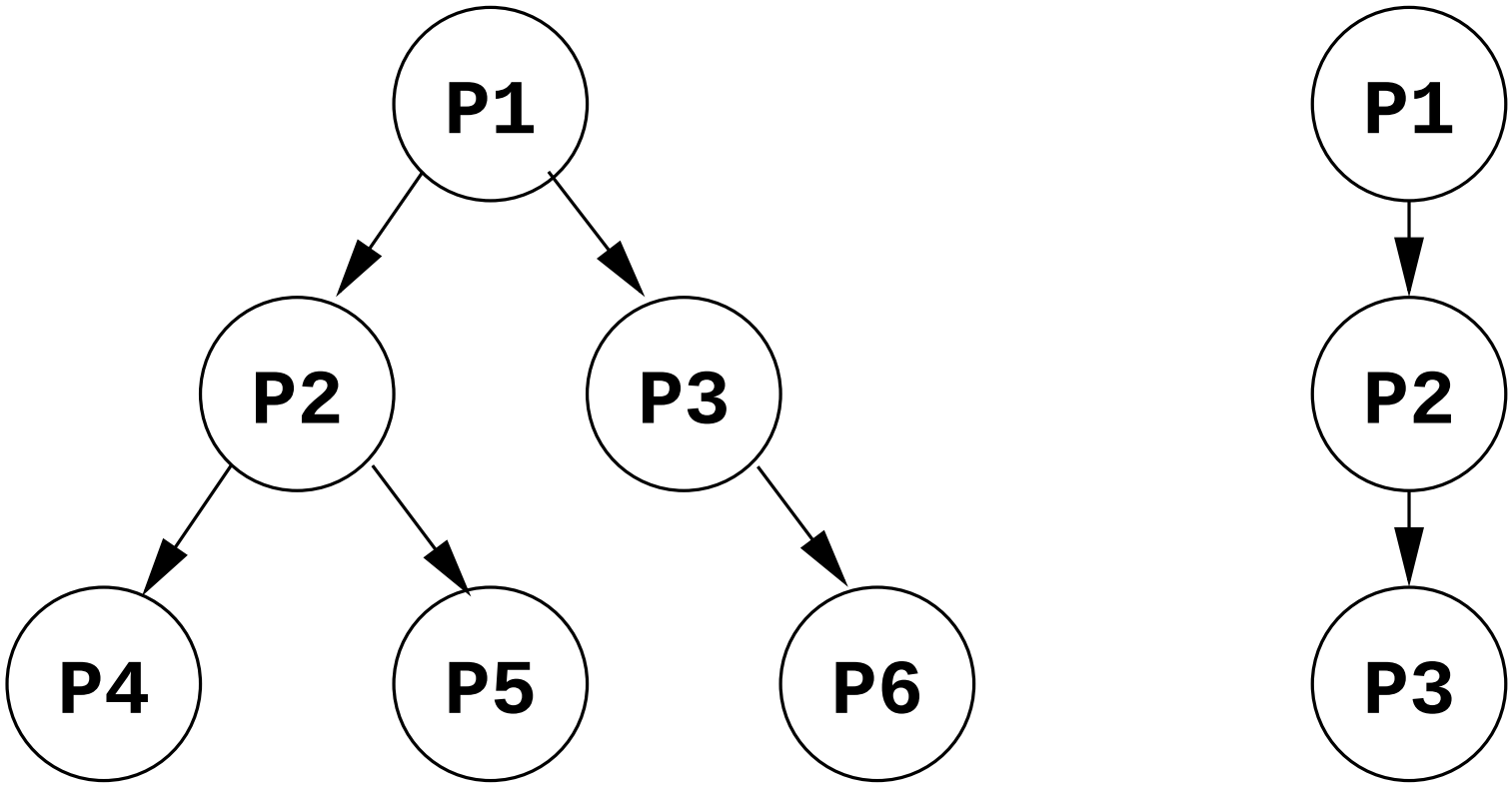
Problema: Como é que se distingue o processo pai do processo filho?

Solução: `fork()` retorna:

- o `pid` do filho ao processo pai;
- `0` ao processo filho.

```
if( (pid = fork()) > 0 ) {
   parent(pid); /* this is executed by the parent
} else if (pid == 0) {
   child();  /* and this by its child */
} else { /* pid == -1 */
   ... /* (parent) handle error */
}
```

Hierarquia de processos em Unix/Linux



- Em Unix/Linux há uma *relação especial*:
  - entre o processo pai e os seus filhos;
  - entre processos que têm um pai comum (*grupo de processos*).

pstree

```
init-+--acpid
     +--atd
     +--battstat-applet
     +--bonobo-activati---{bonobo-activati}
     +--cron
     +--cupsd
     +--2*[dbus-daemon]
     +--dbus-launch
     +--dd
     +--events/0
     +--firefox-bin-+-acroread
     |              +-7*[{firefox-bin}]
     +--gconfd-2
     +--gdm---gdm-+-Xorg
     |            +-x-session-manag-+-gnome-cups-icon
     |                               +-gnome-panel---{gnome-pan
     |                               +-gnome-terminal-+-6*[bash
     |                               |                +-bash---
   [...]                           [...]             [...]
```

Criação de processos: Alternativa a `fork()`

Alternativa: Criar um processo do "nada".

Problema: Há muitos parâmetros a especificar.

- A API do Windows inclui uma chamada ao sistema com 10 argumentos:

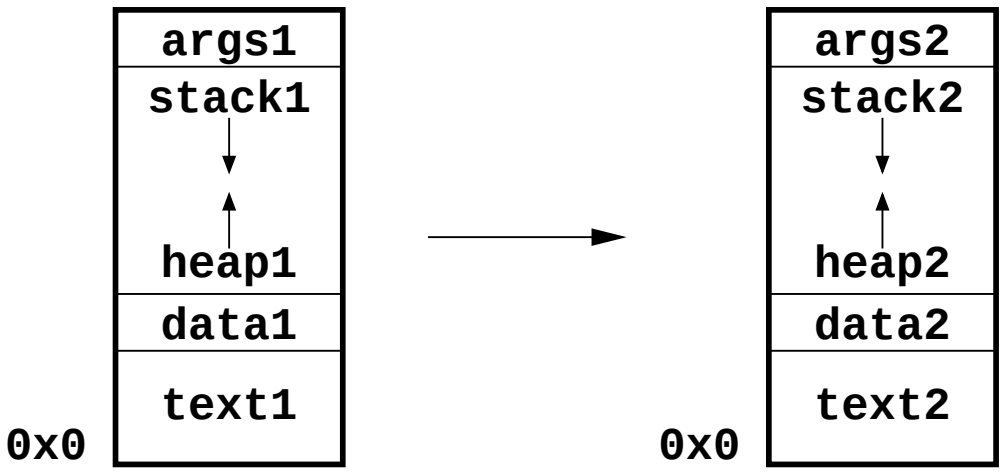
```
BOOL WINAPI CreateProcess(
 _In_opt_    LPCTSTR                lpApplicationName,
 _Inout_opt_ LPTSTR                 lpCommandLine,
 _In_opt_    LPSECURITY_ATTRIBUTES  lpProcessAttributes
 _In_opt_    LPSECURITY_ATTRIBUTES  lpThreadAttributes,
 _In_        BOOL                   bInheritHandles,
 _In_        DWORD                  dwCreationFlags,
 _In_opt_    LPVOID                 lpEnvironment,
 _In_opt_    LPCTSTR                lpCurrentDirectory,
 _In_        LPSTARTUPINFO          lpStartupInfo,
 _Out_       LPPROCESS_INFORMATION  lpProcessInformatio
);
```

Execução de programas

Problema: Como é que um processo pode executar um programa diferente do do pai?

Solução: Usando a chamada ao sistema `execve()`

```
#include <unistd.h>
int execve(const char* filename, char *const argv[]
           char *const envp[])
```



- Substitui o programa em execução pelo contido em `filename`;
- `argv` e `envp` permitem especificar os argumentos a passar à função `main()` do programa a executar.

Shells em Unix

```
while (1) {
    char *cmd = getcmd();
    int retval = fork();
    if (retval == 0) {
        // This is the child process
        // Setup the child's process environment
        // E.g., where is standard I/o, how to handle signals
        execve(cmd, .., ...))
        // exec does not return if it succeeds
        printf("ERROR: Could not execute %s\n", cmd);
    } else { // This is the parent process
        // Wait for child to finish
        wait(retval); // retval was returned by fork()
    }
}
```

Sincronização com `wait()`

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

- Suspende a execução do processo até:
  - um processo filho terminar (`wait()`);
  - o processo filho especificado em `pid` terminar (`waitpid()`);

  Pode retornar ainda quando da ocorrência outros eventos (ver *man page*)
- `status` permite obter informação adicional sobre o processo que terminou, incluindo:
  - O evento que causa o retorno
  - Se o retorno tiver sido via `_exit()`, o argumento que lhe foi passado
- `options` controla o modo de funcionamento de `waitpid()`.

Sincronização com `wait()`: exemplo

```
...
int status;
pid_t pid;
...
if( (pid = wait(&status)) != -1 ) {
   if( WIFEXITED(status) != 0 ) {
      printf(``Process %d exited with status %d\n'',
             pid, WEXITSTATUS(status));
   } else {
      printf(``Process %d exited abnormally\n'', pid);
   }
}
```

- `status` é um inteiro que codifica informação relativa ao evento que causou o retorno de `wait()`/`waitpid()`
- `WIFEXITED()` e `WEXITSTATUS()` são macros que permitem descodificar a informação contida em `status`
  - A *man page* lista e explica estas e outras macros

Terminação de Processos

- Um processo pode terminar por várias causas:
  1. decisão do próprio processo, executando a chamada ao sistema apropriada (directa ou indirectamente, p.ex. por retorno de `main()`);
  2. erro causado pelo processo, normalmente devido a um *bug*, p.ex. divisão por zero ou acesso a uma região de memória que não lhe foi atribuída;
  3. decisão de outro processo, executando a chamada ao sistema apropriada (`kill` em `POSIX`);
  4. decisão do SO (falta de recursos).
- Note-se que um processo pode terminar voluntáriamente quando detecta um erro, p.ex. um compilador não pode compilar um ficheiro que não existe.

Mais chamadas ao sistema

```
#include <unistd.h>
void exit(int status)
void _exit(int status)
pid_t getpid()
pid_t getppid()
```

- `_exit()` termina o processo que a invoca;
- `exit()` é uma função da biblioteca C (deve usar-se com `#include <stdlib.h>`):
  - invoca as funções registadas usando `at_exit()`;
  - invoca a chamada ao sistema `_exit()`.
- Após retornar de `main()`, um processo invoca `exit()`.
- `getpid()` retorna o `pid` do processo que o invoca.
- `getppid()` retorna o `pid` do processo pai.

Roadmap: The Process Abstraction

Further Reading

- Arpaci-Dusseau & Arpaci Dusseau, OSTEP, Ch. 4
- José Alves Marques e outros, *Sistemas Operativos*, FCA - Editora Informática, 2009, Sections 3.1, 3.2,
- A. Tanenbaum, *Modern Operating Systems*, 2nd Ed., Sections 2 e 2.1
- Silberschatz et al *Operating System Concepts*, 7th Ed., Sections 3.1, 3.3 e 3.2.