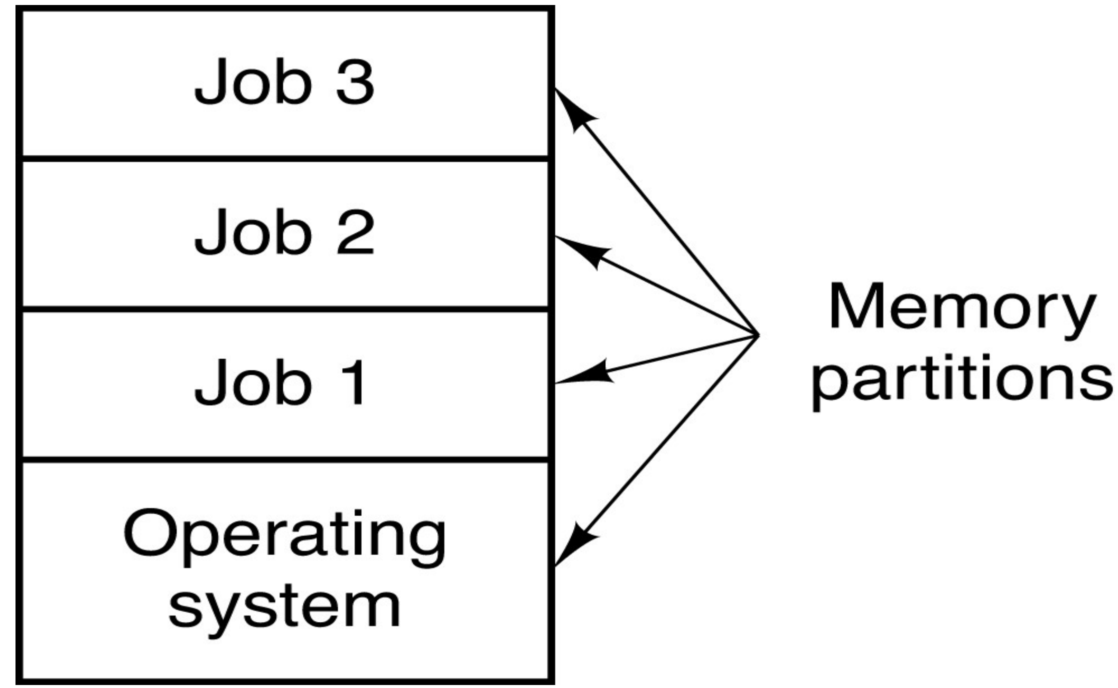
Operating Systems Lab
CSCE 000/3402

Lab Lecture 2: The Fork System Call
Implementing a Shell

Karim Sobh
kmsobh@aucegypt.edu

Multiprogramming

- The OS is loaded into memory simultaneously with other jobs.
- **Multiplexed resources among jobs:**
  - A job yields the CPU upon I/O.
  - OS assigns another job to CPU.
  - Next job needs to be already loaded in memory.
- **Protections techniques:**
  - Protect jobs from each other.
  - Protect the OS from other jobs.
- Transparent job submission and results collection.
- **Non-preemptive**→not interactive.
- Systems are referred to as **minicomputers**.



- **Definition of concurrency:** jobs are considered concurrent by being loaded simultaneously in memory.

Note: *in a uniprocessor environment only one job can have the CPU at any point in time*

Job Scheduling

- Deciding on which job runs next.
- Different scheduling algorithms are available.
- Scheduling types:
  - **Non-preemptive Scheduling:**
    - The job leaves the CPU only at its own will.
    - No mechanism for the OS to kick out a running job from the CPU forcefully.
  - **Preemptive Scheduling:**
    - CPU time can be sliced and shared evenly among different jobs.
    - A mechanism for giving the OS control to kick out a running job from the CPU.
    - Needs hardware support; timers and interrupts.
    - Introduced the notion of a **Timeshare** system.

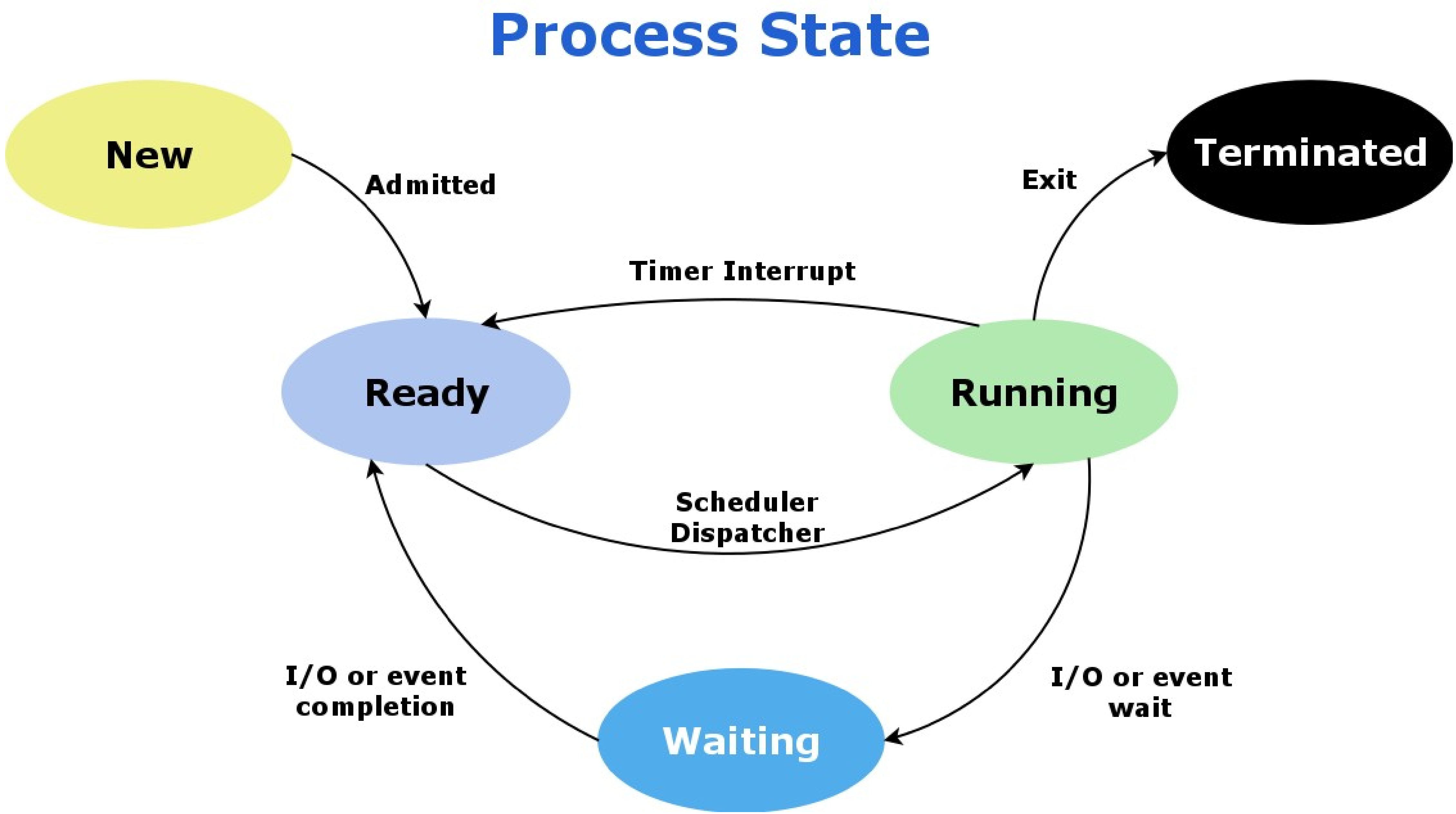
Timesharing

- Timesharing is a variant of multiprogramming.
- Needed for quick response time.
- Based on users with terminals trying to access shared resources.
- Built for interactive processing based on tasks that can be split into short duration transactions.
- Basically I/O was introduced in the equation.
- The seed for utility computing; like the electrical grid.

Process Creation

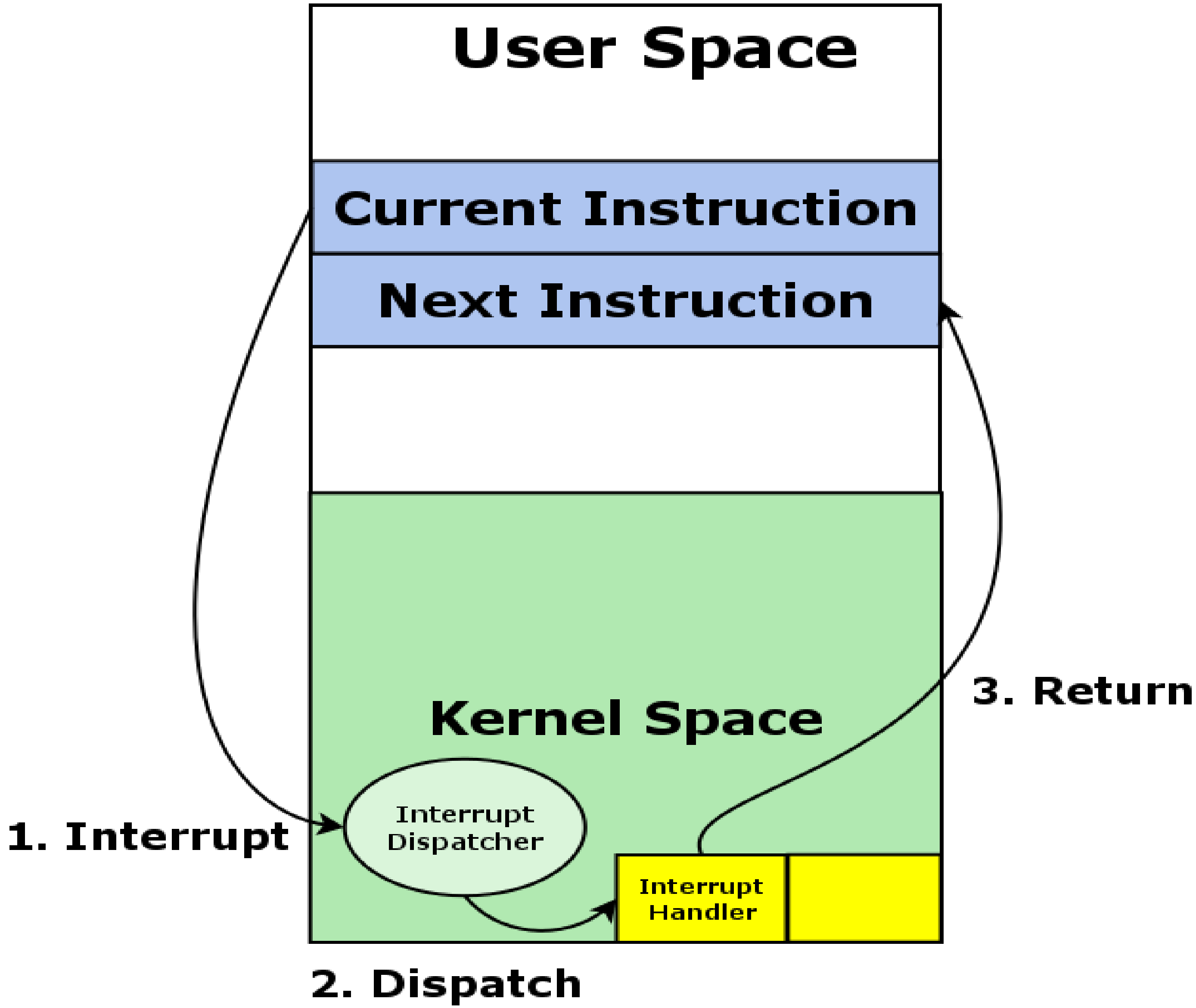
- Four principle events cause processes to be created:
  1. System initialization.
  2. Execution of a process-creation system call by a running process.
  3. A user request to create a new process.
  4. Initiation of a batch job; applies to main frame when different users submit batch jobs and the OS starts one after the other.
- Usually a global parent process is created and the rest of processes are its children forming a hierarchy; e.g. UNIX init process started at boot time.
- The mechanism of creating the process address space differs from one OS to another; e.g. **Copy-on-Write → more on this later**.

Process States

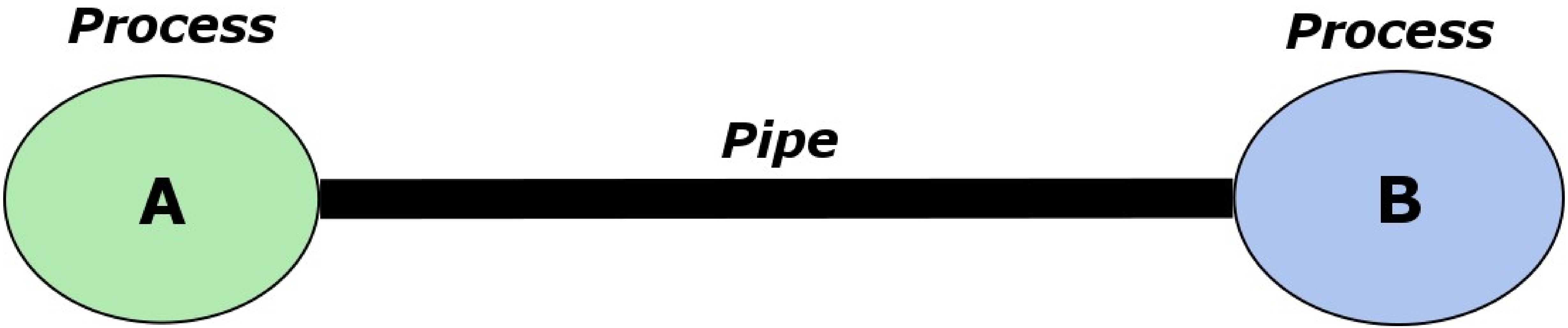


I/O Interrupts

- A program running in the user space requests I/O.
- A software interrupt is generated through an invocation of a system call.
- Execution mode is switched to kernel space.
- Kernel dispatch the interrupt to the interrupt handler.
- After servicing the interrupt execution mode is switched back to user mode.
- Continue from the next instruction in the user program after the instruction that initiated the I/O.



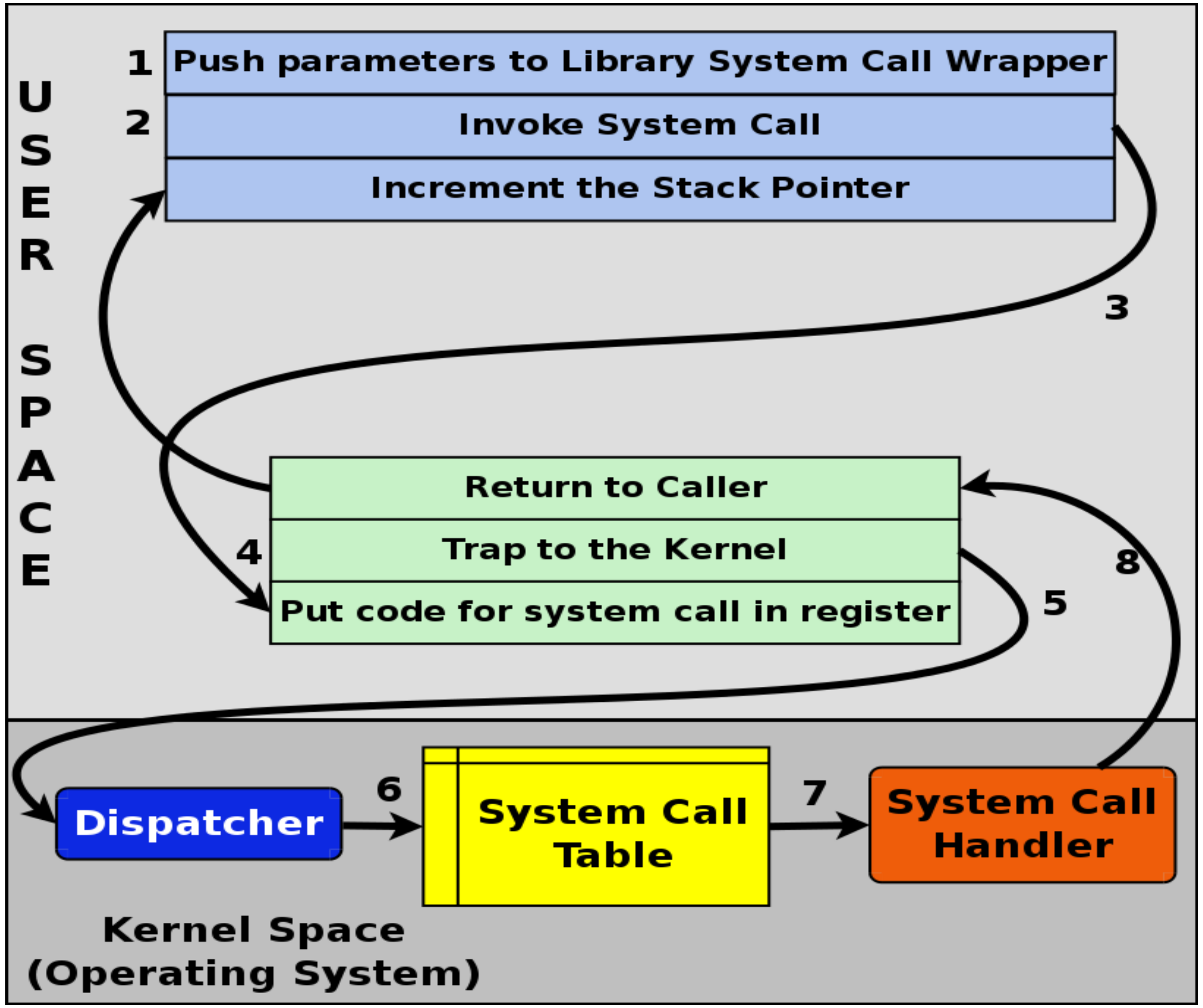
Pipes



- A mechanism for interprocess communication over files.
- Pipes are a sort of pseudo-files that can connect two processes.
- Processes can exchange data/messages over pipes.
- Pipes need to be set in advance.

System Calls

- A well defined **entry point** for user programs **to kernel mode** and allows invoking kernel routines.
- A system call is basically a **function wrapper**.
- Implemented as user mode **library routine**.
- User programs invoke library routines and pass parameters to them.
- A library routine initiates a **software interrupt**; trap.
- Pass arguments to the kernel.
- Kernel execute privileged task.
- Return to the Library and consequently to user program.



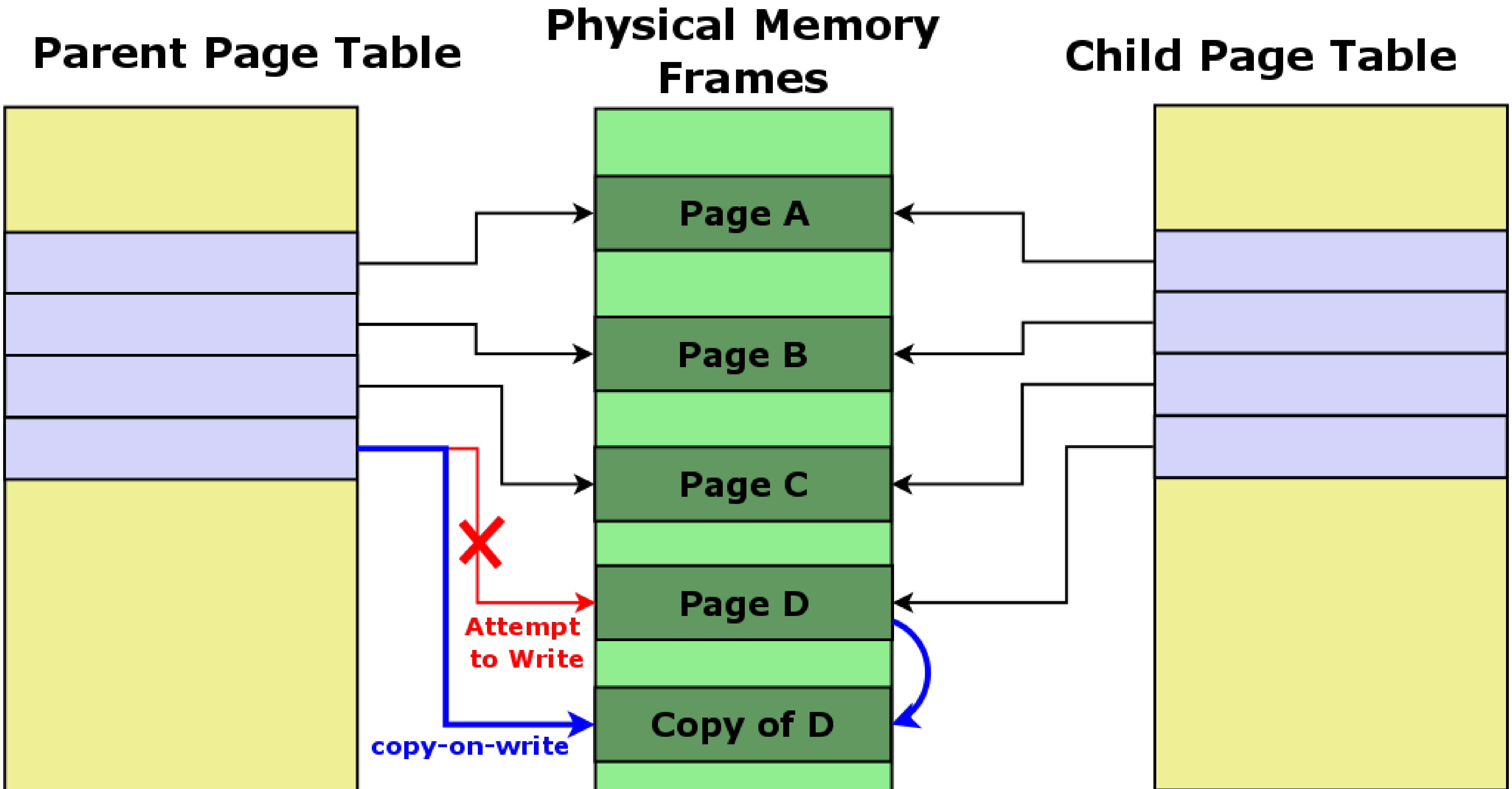
System Call Examples

- Process Management:
  - pid = fork()
  - pid = waitpid(pid,&statloc,options)
  - s = execve(name,argv,environp)
  - exit(status)
- File Management:
  - fd = open(file,how,...)
  - s = close(fd)
  - n = read (fd,buffer,nbytes)
  - N = write (fd,buffer,nbytes)
  - position = lseek (fd,offset,whence)
  - s = stat(name,&buf)
- Directory and Filesystem management:
  - s = mkdir(name,mode)
  - s = rmdir(name)
  - s = link(name1,name2)
  - s = unlink(name)
  - s = mount(name,flag)
  - s = umount(name)
- Miscellaneous:
  - s = chdir(dirname)
  - s = chmod(name,mode)
  - s = kill(pid,signal)
  - seconds = time(&seconds)

Shared Pages (fork system call → copy-on-write)

- Sharing data is tricker than sharing code, but not impossible.
- In UNIX, after a fork system call the parent and the child are required to share both program text and data.
- Each process will have its own page table pointing initially to the same page frames.
- All pages are marked as read-only in the PTE protection bits.
- As soon as the child or the parent start writing, a trap will be generated due to the read-only protection violation.
- The trap handler will:
    - Make a copy of the page that generated the trap.
    - Set the page entry protection bits to READ/WRITE.
    - Link the page table entry of the process causing the trap to point to the new page.

Shared Pages (fork system call → copy-on-write)



The Shell

```c
size_t read_command(char *cmd) {
	if(!fgets(cmd, BUFFER_LEN, stdin))  //get command and put it in line
	return 0;              //if user hits CTRL+D break
	size_t length = strlen(cmd); // get command length
	if (cmd[length - 1] == '\n') cmd[length - 1] = '\0'; // clear new line
	return strlen(cmd); // return length of the command read
}

int build_args(char * cmd, char ** argv) {
	char *token; //split command into separate strings
	token = strtok(cmd," ");
	int i=0;
	while(token!=NULL){// loop for all tokens
	argv[i]=token; // store token
	token = strtok(NULL," "); // get next token
	i++; // increment number of tokens
	}
	argv[i]=NULL;   //set last value to NULL for execvp
	return i;     // return number of tokens
}

void set_program_path (char * path, char * bin, char * prog){
	memset (path,0,1024); // intialize buffer
	strcpy(path, bin);          //copy /bin/ to file path
	strcat(path, prog);          //add program to path
	for(int i=0; i<strlen(path); i++)    //delete newline
		if(path[i]=='\n') path[i]='\0';
}
```

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

#define BUFFER_LEN 1024

int main(){
	char line[BUFFER_LEN];  //get command line
	char* argv[100];        //user command
	char* bin= "/bin/";     //set path at bin
	char path[1024];        //full file path
	int argc;               //arg count
	while(1){
		printf("My shell>> ");                //print shell prompt
		if (read_command(line) == 0 )
			{printf("\n"); break;} // CRTL+D pressed
		if (strcmp(line, "exit") == 0)  break;   //exit
		argc  = build_args (line,argv); // build program argument
		set_program_path (path,bin,argv[0]); // set program full path
		int pid= fork();              //fork child
		if(pid==0){              //Child
			execve(path,argv,0); // if failed process is not replaced
			// then print error message
			fprintf(stderr, "Child process could not do execve\n");
		}else wait(NULL);              //Parent
	}
	return 0;
}
```