Process Management I

Operating Systems

Oleg Goldshmidt

`ogoldshmidt@computer.org`

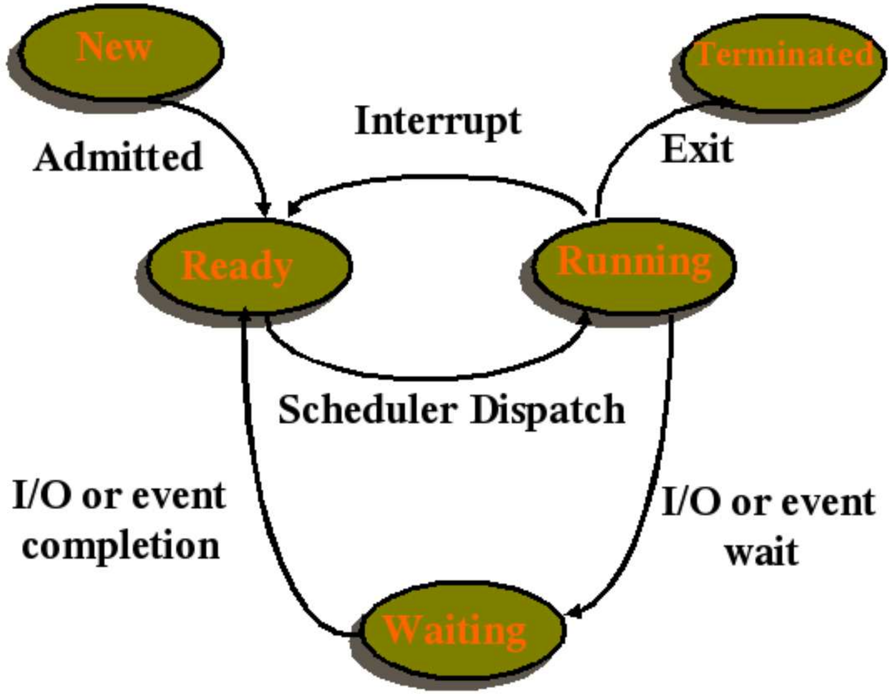
Lecture 2

Process Concept

- a process is a dynamic entity — an instance of a program in execution
  - as opposed to the static concept of a program — a set of instructions (usually in a file on a disk)
  - process execution is sequential (assuming single CPU) — one instruction at a time
- principal components of a process:
  - the program (a.k.a. “text section”)
  - program counter
  - CPU register values
  - stack (function args, local vars, return addresses)
  - “data section” (global variables)
- a program may run several processes

Process States

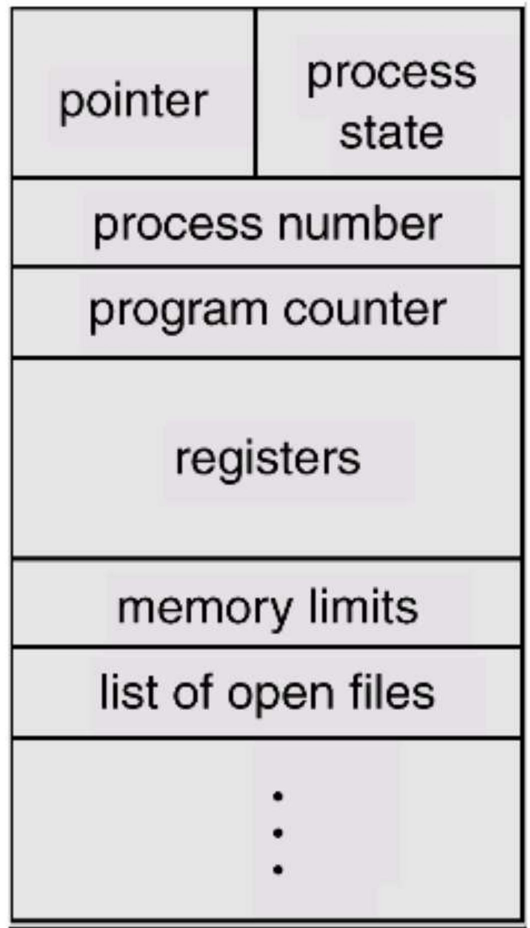
- new — the process is being created
- running — instructions are being executed
- waiting — waiting for some event to occur
- ready — waiting to be assigned to CPU
- terminated — has finished execution



Process Control Block (PCB)

representation of a process

- process state
- program counter
- CPU registers
- CPU scheduling information — priority, queues, parameters
- memory information
- accounting information — CPU and real time usage, time limits, statistics, etc.
- I/O state information — devices, files used, etc.

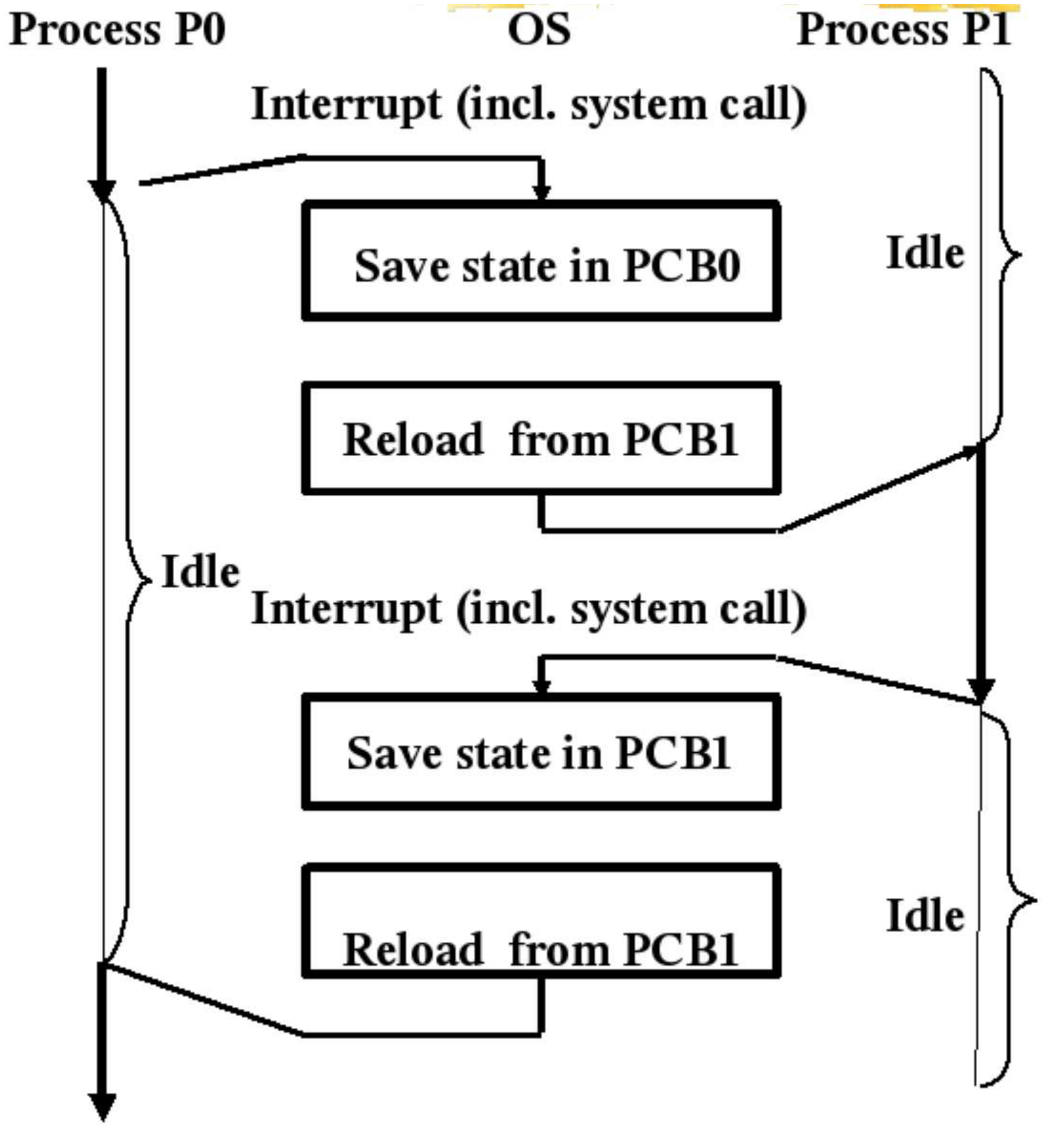


Linux Implementation

in `include/linux/sched.h` (some fields only):

```
struct task_struct {
    volatile long state;
    int prio, static_prio;
    prio_array_t *array;
    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    pid_t pid;
    struct task_struct *parent;
    struct list_head children;
    struct list_head sibling;
    cputime_t utime, stime;
    uid_t uid,euid,suid,fsuid;
    wait_queue_t *io_wait;
};
```

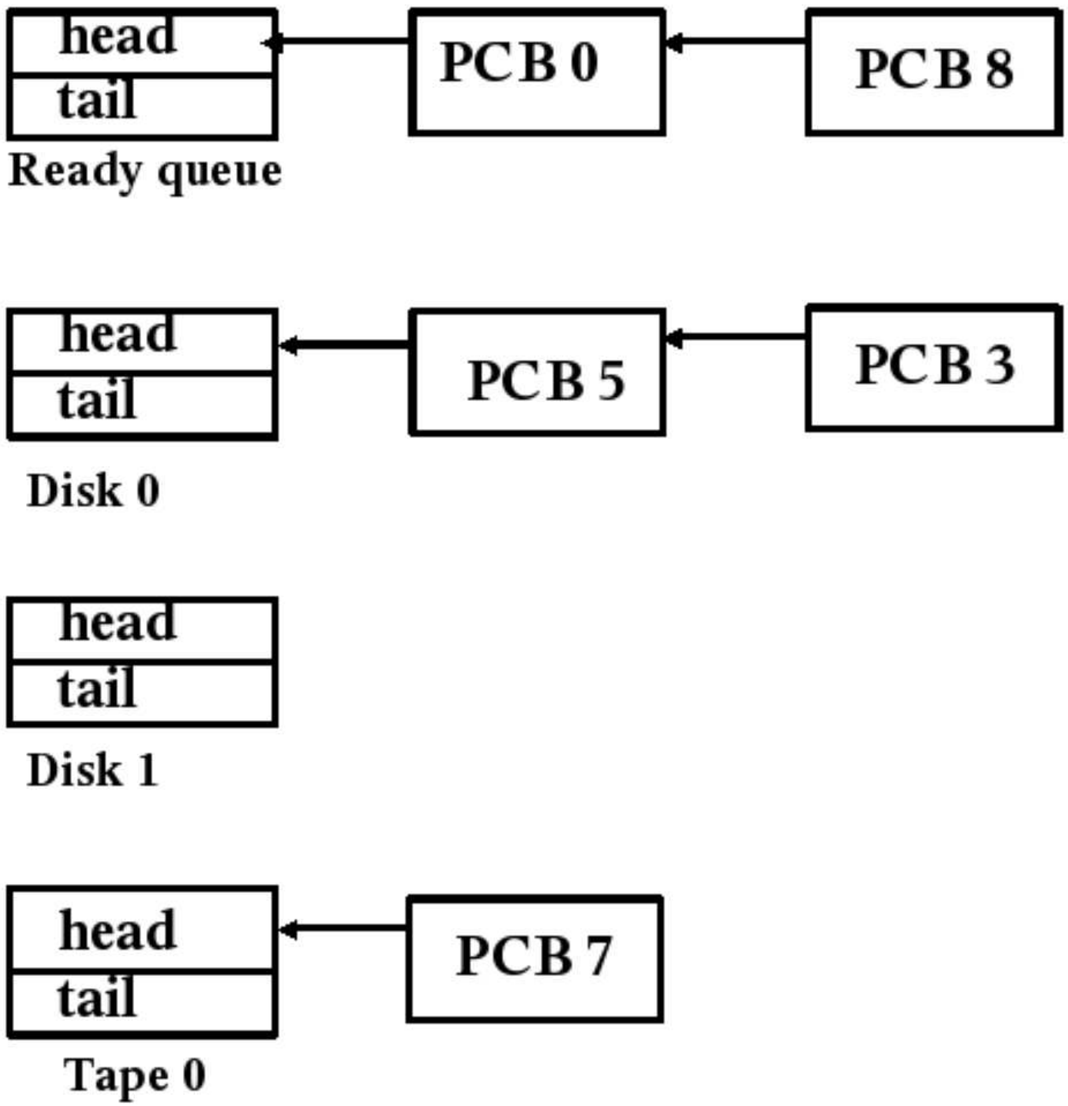
Context Switch I



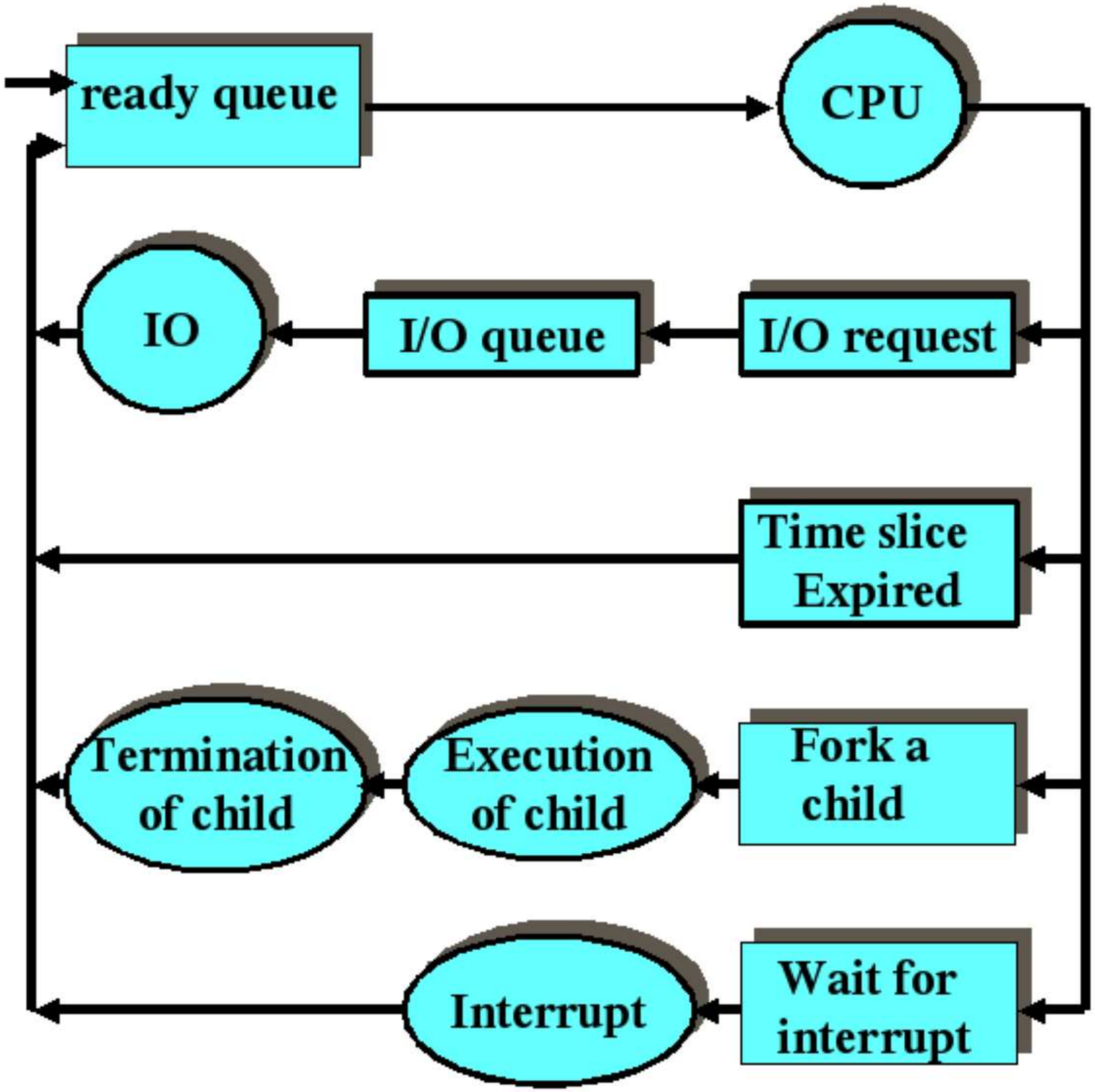
Context Switch II

- save registers of the current process
  - general purpose registers, memory management info, stack pointer
- save PSW of the current process
  - Program Status Word — a CPU register containing execution control bits (e.g., user or kernel mode)
- insert the current PCB into the relevant queue
- mark PCB of the new process as running
- load PSW and PCB of the new process
- new process continues to run from the point where it stopped
- context switch is pure overhead, not useful work!

Scheduling Queues



Process Scheduling



Schedulers and Dispatchers

- short term scheduler
  - moves processes between states
- long term scheduler
  - loads processes from disk to memory
  - controls process mix (CPU-bound vs. I/O-bound)
  - controls the degree of multiprogramming
- mid-term scheduler
  - swaps processes in and out of memory
  - controls process mix
- dispatcher
  - switching context, switching to user mode
  - jumping to proper location in the program

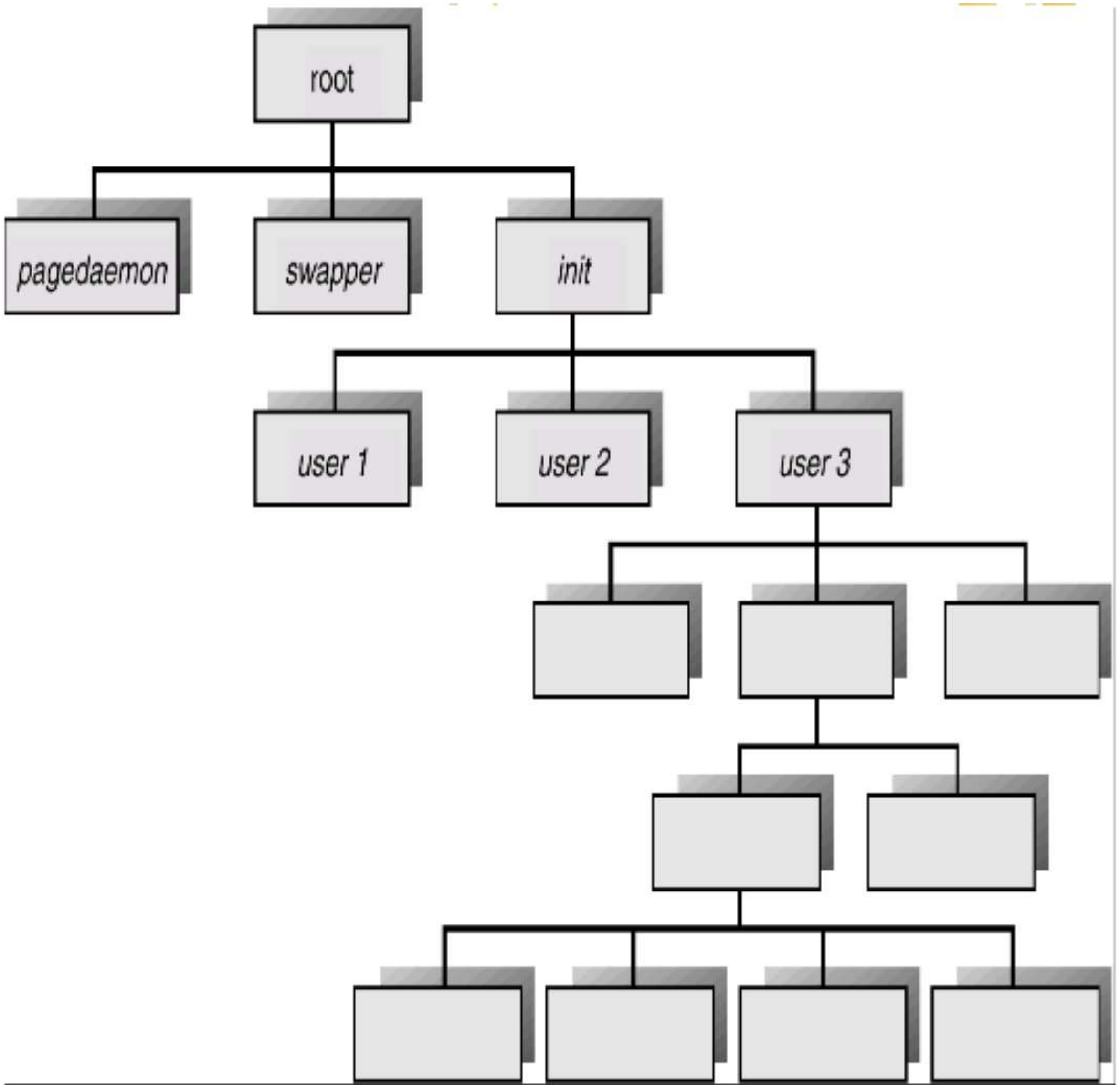
Process Creation I

- processes are created via a system call
  - POSIX: `fork(2)`, Windows: `CreateProcess()`
  - new process gets resources from parent (resource sharing) or from OS
    - sharing parent's resources prevents overloading
  - initialization data: input, environment
- process creation policies
  - execution policy
    - parent continues to execute concurrently
    - parent waits till children (some or all) terminate
  - generation policy
    - child is a duplicate of its father (POSIX)
    - child has a new program loaded into it (Windows)

Process Creation II

- processes are created by other processes, with a few exceptions
- special processes in UNIX
  - swapper/scheduler (`pid = 0`) — system process
  - init (`pid = 1`)
    - normal user process
    - brings up the system
    - all other processes are created via a series of `fork(2)` calls originating in `init`
    - foster parent for orphan processes
  - pagedaemon (`pid = 2`) — virtual memory paging support, system process

UNIX Process Tree



Process Creation: POSIX

- `pid = fork();`
  - `pid_t pid` is a unique integer identifying a new process
- `fork(2)` returns `0` for the child process
  - `pid = 0` for swapper, hence no confusion
  - child can call `getppid(2)` to locate its parent
  - -1 is returned on error
- `fork(2)` creates a copy of the parent process and environment — PCB, links, etc.
  - modern implementations use COW: resources are allocated, but no copy occurs unless something is changed (either in child or in parent)

Process Creation: Example

```
#define _POSIX_SOURCE 1
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int child;
    if ((child = fork()) < 0) {
        perror("fork error");
        exit(EXIT_FAILURE);
    } else if (child == 0) {
        printf("child: PID %d\n",getpid());
    } else {
        printf("parent: child PID %d\n",child);
    }
    exit(EXIT_SUCCESS);
}
```

Executing A New Program I

- `fork(2)` creates a copy of the parent process — how can we make a child execute a different program?
- the `exec()` family of system calls: `execl(2)`, `execv(2)`, `execle(2)`, `execve(2)`, `execlp(2)`, `execvp(2)`
  - often referred to, collectively, as “exec”
- replace the process virtual memory space with a new program by loading an executable file into memory
- the first 4 take a path argument, the last 2 take a filename argument (`PATH` is used to search for the executable)

Executing A New Program II

- the `l` functions (`execl()`, `execle()`, `execlp()`) require a list of arguments terminated by a null pointer
- the `v` functions (`execv()`, `execve()`, `execvp()`) require building a vector of pointers to arguments
- the `e` functions (`execle()`, `execve()`) allow to pass environment (the others inherit environment from parent)
- this is how, e.g., shells work
- more info during the drill sessions

Executing A New Program: Example

```
char *env[] = {"USER=oleg","PATH=/bin",NULL};
pid_t pid;
if ((pid = fork()) < 0) {
    perror("fork error");
    exit(EXIT_FAILURE);
} else if (pid == 0) {
    if (execle("/bin/echo","echo",
               "arg1","arg2",(char*)0,
               env) < 0) {
        perror("execle error");
        exit(EXIT_FAILURE);
    }
}
/* the rest of the parent code */
```

Creating Processes: Windows

- not POSIX — no `fork(2)` or `exec(2)`
- `CreateProcess()` is a combination of `fork()` and `exec()`
  - some differences in building the command line, parsing
- `fork(2)` without `exec(2)` is more difficult
  - useful when the child needs to inherit some resources of the parent
  - handles to objects are inherited explicitly, via a variety of means
- no process hierarchy or a global concept of parent, but children can be grouped if the parent has been created with a particular attribute

Process Creation Overhead I

- find some files and do something with each of them
- `find . -exec ls -ld \{\} \;`
  - creates a process per file
- `` ls -ld `find .` ``
  - this will work, unless the output of `find .` is too long to be used on the command line
- `find . | xargs ls -ld`
  - `xargs` takes arguments from `stdin` and passes up to `ARG_MAX` arguments to command at a time — creates only a few processes

Process Creation Overhead II

in `/usr/src/linux-2.4.21-27.0.4.EL`:

```
# time nice find . -exec ls -ld \{\} \;
real     4m55.845s
user     0m14.990s
sys      0m28.290s
# time nice ls -l ‘find .‘
bash: /bin/nice: Argument list too long
real     0m1.542s
user     0m0.810s
sys      0m0.040s
# time nice find . -print0 | xargs -0 ls -ld
real     1m6.878s
user     0m0.990s
sys      0m0.580s
```

Normal Process Termination

- a process can be killed by itself, some other process, or the kernel
- normal termination — `exit(3)`
  - may return data to parent
  - returning from `main()` is equivalent to `exit(3)`
  - all resources are deallocated
    - call all handlers registered with `atexit(3)`
    - close all standard I/O streams, etc.
    - release memory
  - `_exit(2)` is called by `exit(3)` to take care of OS-specific details

Abnormal Process Termination

- abnormal termination — `abort(3)`
  - invoked by another process, typically parent
  - calling process needs to know the `pid` (`fork(2)` returns the child's `pid` to the parent)
  - a special case of a signal — `SIGABRT`
  - reasons:
    - resource usage exceeded
    - task no loner needed,
    - parent is exiting (cascading termination)

Process Termination: Details I

- parent must be notified
  - for normal termination `exit(3)` or `_exit(2)` are called with an “exit status” argument
    - “exit status” is converted to “termination status” by the kernel when `_exit(2)` is called
  - for abnormal termination the kernel generates the “termination status”
  - parent can obtain the termination status using `wait(2)` or `waitpid(2)` (handler for `SIGCHLD`)

Process Termination: Details II

- what if parent terminates before child?
  - cascading termination (VMS) — child cannot exist if parent is terminated (normally or abnormally)
  - UNIX — every process has a parent, if parent terminates the kernel changes the `ppid` to `1` (`init`) for all the children
- what if a child terminates before the parent?
  - the kernel must keep minimal information (`pid`, status, usage statistics) for every terminated process
  - the process becomes a “zombie”
  - `init` periodically calls one of the `wait()` functions to prevent clogging by “zombies”

Terminating Processes: Windows

- TerminateProcess() is not a substitute for kill(2)
  - not all DLL's call their exit routines
  - use only in extreme circumstances
  - use WM_CLOSE message instead
- remember: no concept of parent
  - if the parent was created with CREATE_NEW_PROCESS_GROUP flag set, the children can be grouped
  - GenerateConsoleCtrlEvt() can send Control-C or Control-Break signals to the group
  - only the children who share a console with parent will receive the signal