Week 10

Processes (review)

1/11

Process = executing program (state = registers(PC), heap, stack, ...)

Linux/Unix processes identified by a unique process ID (*pid*)

`pid_t fork()` ... clone a process

- returns child pid to *parent*; returns 0 to *child*

`pid_t getpid()` ... returns pid of current process

`pid_t wait(int *status)` ... wait for child process

- returns on child state change e.g. exiting, sending signal, ...

`int kill(pid_t pid, int sig)` ... send signal

- send a signal to a specific process or process group

Process-related System Calls

2/11

`int execve(char *Path, char *Argv[], char *Envp[])`

- replaces current process by executing `name` object
  - *`Path`* must be an executable, binary or script (starting with `#!`)
- passes arrays of strings to new process
  - both arrays terminated by a `NULL` pointer element
  - *`envp`*`[]` contains strings of the form `key=value`
- much of the state of the original process is lost, e.g.
  - a new virtual address space is created, signal handlers reset, ...
- new process inherits open file descriptors from original process
- on error, returns -1 and sets `errno`
- if successful, does not return

Exercise 1: Executor

3/11

Write a small program that will run other programs

- reads, one per line, values for command-line arguments
- trims each line and stores pointer to it in array `args[]`
- uses `args[0]` as the path of the program to run
- uses `args[]` as `argv[]` in the exec'd process
- passes no `envp[]` values (i.e. `envp[0]=NULL`)
- invokes the specified program then waits for it to complete
- displays the exit status of the invoked process

... Process-related System Calls

4/11

`int kill(pid_t ProcID, int SigID)`

- send signal *`SigID`* to process *`ProcID`*
- various signals (POSIX) e.g.
  - `SIGHUP` ... hangup detected controlling terminal/process
  - `SIGINT` ... interrupt from keyboard
  - `SIGILL` ... illegal instruction
  - `SIGFPE` ... floating point exception (e.g. divide by zero)
  - `SIGKILL` ... kill signal (e.g. `kill -9`)
  - `SIGSEGV` ... invalid memory reference
  - `SIGPIPE` ... broken pipe (no processes reading from pipe)
- on error, returns -1 and sets `errno`
- if successful, returns 0

Signals

5/11

Signals can be generated from multiple sources

- from a program via `kill()`
- from the operating system (e.g. timer)
- from a device (e.g. i/o)

Processes can define how they want to handle signals

- using **`signal(int SigID, sighandler_t Handler)`**
- *`Handler`* can be `SIG_IGN`, `SIG_DFL` or a function
- *`SigID`* is one of the OS-defined signals

Interrupts

6/11

Interrupts are signals which

- cause normal process execution to be suspended
- a *handler* then carries out tasks related to interrupt
- control is then returned to the original process

Example (with a single process):

- process starts some disk i/o (e.g. read a block of data)
- then carries out major in-memory computations
- when data fetched from disk, process is interrupted
- data is placed in a buffer for later access by process
- in-memory computation resumes

... Interrupts

7/11

Interrupts are frequently associated with input/output

- in-memory computations are very fast (*ns*)
- input/output operations are very slow (*ms*)

One way to handle i/o ...

- process invokes i/o operation (e.g. write to disk)
- then waits for operation to complete (e.g. 100*ms*)
- once data is written, process continues

Downside: wastes a *lot* of time, reduces system throughput

Solution: run another process "while you wait"

Multi-tasking

8/11

Multi-tasking = multiple processes are "active" at the same time

- processes are not necessarily *executing* simultaneously
  - although this could happen if there are multiple CPUs
- more likely, have a mixture of processes
  - some are *blocked* waiting on a signal (e.g. i/o completion)
  - some are *runnable* (ready to execute)
  - one is running (on each CPU)

Aims to give the appearance of multiple simultaneous processes

- by switching between them after each runs for a defined *time slice*
- after timer counts down, current process is *pre-empted*
- a new process is selected to run by the system *scheduler*

Scheduling

9/11

Scheduling = selecting which process should run next

- processes are organised into *priority queue(s)*
  - where "highest" priority process is always at head of queue
- priority determined by multiple factors, e.g.
  - system processes have higher priority than user processes
  - longer-running processes might have lower priority
  - memory-intensive processes might have lower priority
  - processes suggest their own priority (*nice*-ness)

Unix/Linux processes have priority values in range 0..139

- 0 is highest priority (system processes); 100+ are user-process priorities

---

... Scheduling

10/11

Abstract view of the OS scheduler

```
onTimerInterrupt()
{
   save state of currently executing process
   newPID = dequeue(runnableProcesses)
   setup state of newPID (e.g. load pages)
   transfer control to newPID (i.e. set PC)
}
```

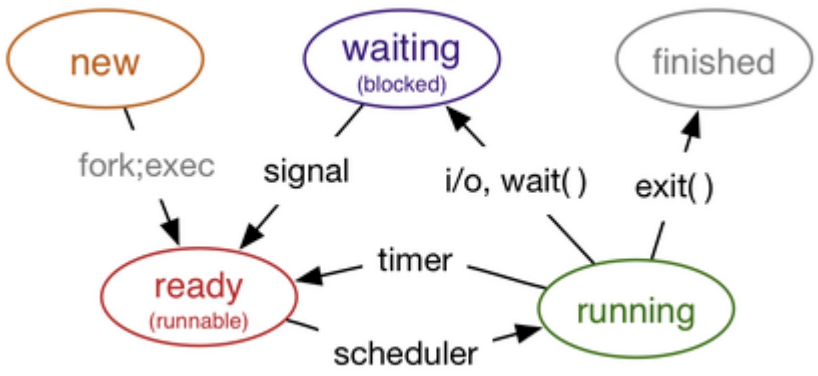
Process information stored in *process control block*

- process identification data (e.g. pid, ppid, pgid, uid, gid, ...)
- process state data (e.g. registers, stack, heap, page table, ...)
- process control data (e.g. scheduling state/priority, open files, ...)

Process States

11/11

How process state changes during execution



Produced: 5 Oct 2017