Operating Systems

Recitation 6

Plan

- Multitasking
  - Single process
  - Cooperative multitasking
  - Preemptive multitasking
- Processes API (Linux)

What is a process?

- Process == task
- Execution of an assignment defined by a program
  - Process is instance of program execution
  - Two opened notepads == two processes
  - But not straightforward - several Firefox windows may be supported by single process

What is a process?

- For OS
  - “Context of execution”
  - independent entity consuming resources (CPU time, memory, disk)

- How does the OS switch between processes?

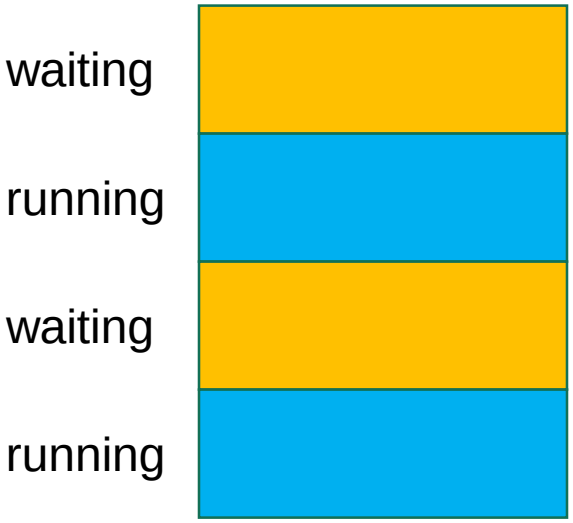
Single-process system

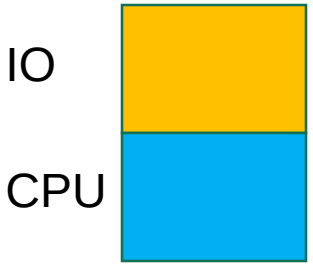
- Running program gets full control over computer resources
- Next program starts after previous ended

- What's the problem?

Single-process system

- Not efficient
  - CPU idle a lot of the time – for example when waiting for printer to finish
  - May want to prioritize





Cooperative Multitasking

Programs voluntarily give up CPU time for other programs

- Program gets control only when other program “yields”
- User may select/activate program to run
  - Pause editor and look into spreadsheet

- Examples:
  - Windows 3.1, old Mac OS
- Today not in common use. Why?

Cooperative Multitasking

Difficult for programmers to write a program aware of other programs

- How does a program know when to yield CPU to other program, and when (if ever) will it resume?
- We may need access to private memory of other processes
  - To know when and how to yield…
  - Lots of IPC

Preemptive multitasking

OS *Scheduler* decides when to run each process and for how long

- CPU time divided to slices
- Process gets fair share of CPU time (priority & policy)
- OS manages each process state, guards its private memory and manages shared resources (disk, printer,…)

Preemptive multitasking

- Process may enter “waiting state” (e.g. wait for other process/event to complete) so OS won’t allocate any CPU time at all
- Process suspended without its permission (i.e. preemption)
- OS manages queues of prioritized processes

Multitasking: Some Requirements

- To switch between programs, OS able to
  - Interrupt process on intervals
  - Freeze state of paused program (e.g. current instruction, register and memory state)
    - And resume later
  - Use “scheduler” to decide what to do next according to some policy

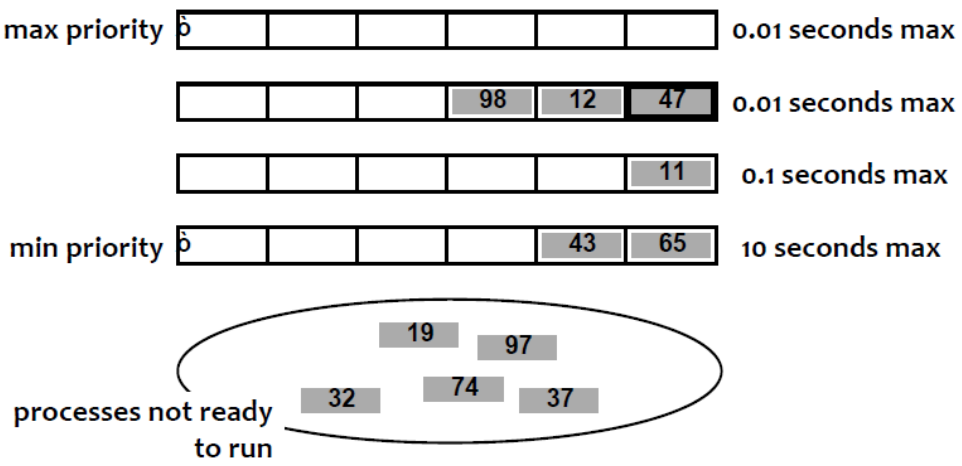
Process descriptor

Each process has a “descriptor” (defined in `struct task_struct` in `/usr/src/linux-headers-XXX/include/linux/sched.h`):

- Process Identifier (`pid`)
- Program counter - Address in memory of next instruction to execute
- Context data - Registers
- Statistics - CPU time used till now
- State - Running, ready, waiting, …
- Priority - Low, high
- Open files
- Memory map

Scheduling

- In class saw example of well-known scheduler implementation – Multilevel feedback queues
- A Unix OS manages several queues of process descriptors
  - Doubly linked list, Priorities



מעבר בין תורים

- מעבר בין תורים:
  - תהליך שצבר זמן ריצה רב עובר לתור נמוך יותר
  - תהליך שלא רץ זמן רב עובר לתור גבוה יותר, למניעת הרעבה
  - תהליך שחיכה למשאב וקיבל אותו נכנס לתור גבוה מאוד כדי שישחרר את המשאב מהר ככל האפשר
- העדיפות שאנו מעניקים לתהליכים משפיעה על הפרמטרים של מעבר בין תורים ולא קובעת תור הספציפי לתהליך
- קל לממש מדיניות הוגנת ויעילה
- קל לממש יכולות מיוחדות, כגון תהליכים שרצים רק שאף תהליך אחר לא מוכן לריצה

Multitasking API

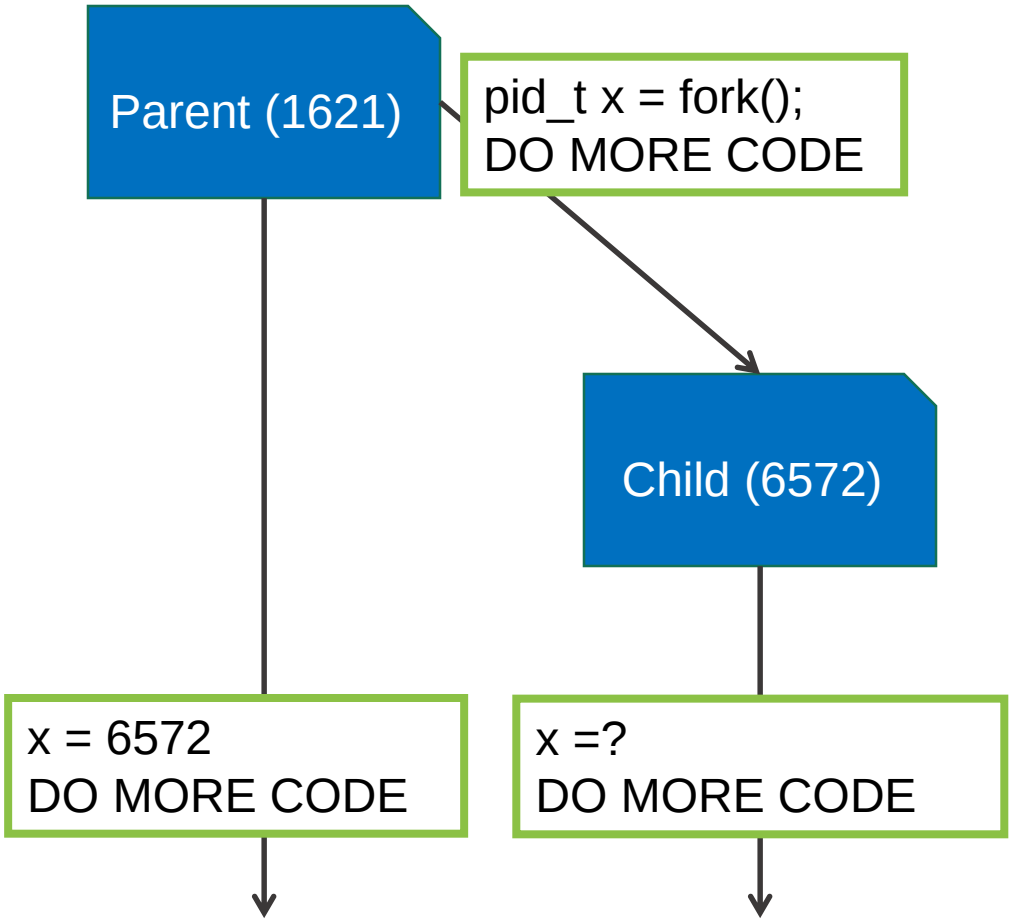
- Process created by `fork()` from “parent” process
- Linux starts with 2 processes
  - `Swapper` (manages memory)
  - `init` (`pid=1`) – all processes are its sons (forked)
- “Parent” can fork multiple times
  - Son too…

fork()

`pid_t fork()` - returns identifier of process

- Copies parent to son (clone)
  - Code and execution point
  - Environment = open files and `fd`'s, current working lib
  - Memory = buffers, variables and values
- Return value
  - Son = 0, parent = son's new pid, error = -1
- Windows? – `CreateProcess` (sort of `fork+execv`)

fork()


Parent (1621)
pid_t x = fork();
DO MORE CODE
Child (6572)
x = 6572
DO MORE CODE
x =?
DO MORE CODE

fork()

- Typical fork usage example

```
int pid = fork();
if (pid < 0)
    // fork() failed – handle error

if (pid == 0)
    // son process – do son code
else
    // parent process – do parent code
```

Peculiarities (1)

- What would this do?

```
main() {
    int pid1 = fork();
    int pid2 = fork();
}
```

- How many processes?

Peculiarities (2)

- What would this output?

```
main() {
    fork();
    fork();
    printf(“hello”);
}
```

- “hellohellohello”?

Peculiarities (3)

- What would this do?

```
main() {
    int pid = fork();
    if (pid) {
        fork();
    }
    fork();
}
```

- How many processes?

execv()

```
int execv(const char *filename, char
*const argv[])
```

- Run a program in same process
  - `filename` - file with program to run
  - `argv` - same as in main (`argv[0] == filename`)
  - Last entry in `argv` should be `NULL`
  - On failure -1, **on success <u>doesn't return</u>!**

execv()

- Typical usage example

```
main() {
  char *argv[] = {“/bin/date”, “-u”, NULL};
  execv(“/bin/date”, argv);
  printf(“hello”);
}
```

- What would this do?

- ***Code example***

`wait()` / `waitpid()`

```
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status,
int options)
```

- Parent can wait for son to end (any son / specific)
  - status - son's status when finished
  - Returns -1 if all sons finished or already `wait()`ed, or **`pid`** of some process that finished

wait() / waitpid()

- How do we wait for all child processes to end?

```
while (wait(&status) != -1);
```

- `pid_t getpid();`
  - Get process id of current process

- ***Code example***

Zombies

- Sometimes need information from son-process, or can't continue until it finishes → need to wait for it
- …if son is finished, and `wait()` not invoked yet, it is a "**Zombie**".
  - Exists only as process record
  - Deleted once son `wait()`ed
- What if parent is finished, and son isn't?
- ***Code example***

Linux Signals

- Inform process async. on some event
- Form of IPC and for messages from OS about relevant events
- 32 types in UNIX
- Some signals are handled automatically by OS
  - SIGKILL, SIGSTOP
- Others handled by signal handler of process
  - OS queues signals (only 1 for every type!)
  - Upon returning to user-mode process handles them using handler
  - Can write our own signal handlers too!

kill()

```
int kill(pid_t pid, int sig);
```

- Officially - signal process to die
  - pid = process id
  - sig = signal to send
- Actually name is misleading
  - can send any signal
  - Real kill signal intercepted by OS…
- **`kill`** utility from command line really does send a `SIGKILL` / `SIGTERM` signal