Operating System: Process

Alex Chi

Update: March 9, 2020

Contents

1 Process

- Process is an abstraction provided by the operating system.
- It is just a **running program**.
- Its data resides on disk or other place.
- OS allocates resource for a process, let the program think itself as is the only program running the computer.

1. From the **memory** perspective
   - `text` section, contains program code
   - `data` contains global variables
   - `stack` contains temporary data
   - `heap` contains memory dynamically allocated
   - On x86 Linux, stack grows downwards from the highest address in user space.
2. Difference from Program
   - Program is just code.
   - It becomes process when executing.
3. Q&A

   (a). Is memory continuous in user-space always continuous physically?

   Nope. User-space memory is mapped through paging table. Behind that the memory might not be continuous.

2 Process State

- new (being created, admitted → ready)
- running (being executed, exit → terminated)
- waiting (running → IO (waiting) → ready
- ready (to be assigned to a processor, scheduler dispatch → running)
- terminated
- You may think of transition of states is a DFA. (refer to figure in slides)
- Side note: terminated process may still reside in memory as parent process is waiting for its return code. In xv6, this kind of process is marked as zombie.

3 Process Control Block

- information associated with each process
- which normally includes:
  - process state
  - process number
  - program counter
  - CPU registers (sp, caller-saved and callee-saved registers)
  - CPU scheduling information
  - memory-management information
  - user account information
  - I/O status information

3.1 xv6

- for example, in xv6 (reference: github.com/mit-pdos/xv6-riscv)
- where `state` is process state
- `pid` is process number
- program counter and registers are saved in `tf` or trapframe
- xv6 uses simple round-robin for scheduling, so there's no specific CPU scheduling information
- `pagetable` saves memory-management information
- xv6 is single user OS
- `ofile` stores all open files

```
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  struct proc *parent;         // Parent process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;               // Bottom of kernel stack for this process
  uint64 sz;                   // Size of process memory (bytes)
  pagetable_t pagetable;       // Page table
  struct trapframe *tf;        // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

3.2 Linux

- doubly-linked-list
- user struct also contains some process information

```
struct task_struct {
  pid_t pid;
  long state;
  unsigned int time_slice;
  struct task_struct *parent; // parent process
  struct list_head children;  // children process
  struct files_struct *files; // list of open files
  struct mm_struct *mm;       // address space
}
```

3.3 Windows

- Executive Process Block (kernel space)
- Kernel Process Block (kernel space)
- Process Environment Block (user space)

4 Context Switch

- [refer to figure in slides]
  - PCB_0 running
  - interrupt or syscall, user → kernel space
  - save state into PCB_0
  - find a runnable process
  - reload state from PCB_1
  - switch back to user space
  - PCB_1 running
- Here I personally give an example of xv6 process switch
  - user program calls `ecall`, or there's a timer interrupt, which jumps to trampoline
  - trap saves user program status into trap frame, loads kernel page table
  - jumps to `usertrap`, process syscall, if is timer interrupt, yield to scheduler kernel thread
  - scheduler picks next process to run
  - calls `usertrapret`, which sets up supervisor-mode registers and jumps to trampoline
  - trampoline recovers user program registers and loads user page table, calls `sret` to go into user mode
  - now new process is running

5 Process Scheduling

- the process scheduler selects among available processes
- scheduler maintains a scheduler queue
  - job queue - set of all processes in system
  - ready queue - all processes residing in memory, ready to execute
  - device queue - set of processes waiting for I/O
  - process migrate among these queues
  - some OS doesn't have job queue

5.1 Ready Queue and Various I/O Device Queues

refer to figure in slides

5.2 Representation of Process Scheduling

also a DFA. refer to figure in slides.

5.3 Schedulers

- long-term scheduler - selects which process to be brought into ready queue
- short-term scheduler - selects which process to run next
- most systems only have short-term scheduler, as all processes are in ready queue
- **short** and **long** indicates invoking interval. Long-term scheduler is invoked every few seconds or minutes
- the long-term scheduler controls the degree of multi-programming
- Process can be classified as:
  - I/O-bound process: e.g. HTTP server, database
  - CPU-bound process: e.g. machine learning, data mining, Bitcoin

1. Q&A
   (a). What is time slice?
   For this time, just assume it is time that a process is allowed to run. For example, 1ms. If a process has been running for 1ms, a timer interrupt will be triggered and it will be put into sleep. (In fact a process is not always guaranteed to have a running time of exactly 1ms, in xv6)

6 Operations on Processes

6.1 Process Creation

- parent process create child process, which in turn forms a process tree

6.1.1 Challenges

1. Resource sharing

- parent and child share all resources (hard to distinguish two process)
- a subset of resources (file handler as stdin & stdout, memory from parent as forking, aka. copy-on-write fork)
- share no resource (abundant resource)

2. Initialization data

   data passed from parent process to child process (IPC, argc & argv, etc.)

3. Execution
   - parent and child execute concurrently
   - parent waits until child terminate (or there'll be zombie process)
4. Address Space
   - child duplicate of parent
   - child has a program loaded into it
5. C Program
   - fork syscall
   - exec syscall
6. fork syscall

```
int main() {
  pid_t pid;
  pid = fork();
  if (pid < 0) {
    // fork failed
    return 1;
  } else if (pid == 0) {
    // child process
    execlp("bin/ls", "ls", NULL);
  } else {
    // parent process
    wait(NULL);
  }
  return 0;
}
```

- fork creates new process
- OS will duplicate memory of original process
- exec will replace current process's memory
- what happens inside OS?

```
li a0, SYS_FORK
ecall # now PC points to instruction next to ecall (ret)
ret
```

- OS create a new process
- OS writes child process pid to `a0` (assume `a0` saves return value) for parent process, and writes `0` to `a0` in child process
- OS schedules either of the two processes
- note that child process's pid is NOT zero, use `getpid()` to get real pid.

7. exec syscall
   - exec syscall is called by a family of functions such as `execvp`, `execlp`.

6.1.2 Q&A

1. Do two process share the same instances of `main()`?
   They run on the same code.
2. Are pids of parent and children randomly assigned?
   `init` has a pid of `1`. All other process are sequentially assigned.
3. Is return value in parent `fork()` the pid of child process?
   Yes.
4. What will happen if parent exits in an abnormal way?
   The child process will become orphan. Then they will be attached to `init` in Linux. After the child process exits, it will become zombie process.
5. Will two process fork and exit individually?
   Yes. You'll never know which process will be scheduled first, or exit first. Adding `wait` syscall will force parent to wait for the children.
6. What about resource sharing?
   Local variable, static variable, global variable, etc. will have the same value after forking. But modification to these variables will not be visible to on another.
   If the OS implements copy-on-write forking, then the program code might be shared.
7. How to know parameters for syscall?
   Type `man fork`, `man wait` in terminal.
8. What about pointers?
   After forking, two pointers have the same logical address, but the same address in different process actually resides in different locations in physical memory. By the way, you'll never have access to parent process in child.

6.2 Process Termination

1. exit syscall
   - last statement of your program. called by glibc in your C program.
2. wait syscall
   - parent get return value of child by `wait` syscall
3. abort
   - parent abort child executing

7 Inter-Process Communication

7.1 Basics

1. Two models of IPC
   - shared memory
   - message passing
   - [refer to figures in slides]
2. Shared Memory
   - talk through shared space
   - fast, as fast as memory speed
   - complicated, manually control concurrency
3. Message Passing
   - use message queue in kernel
   - slow than shared memory
   - can be used across network

7.2 Producer-Consumer Problem

- **unbounded-buffer** no limit on size of buffer
- **bounded-buffer** fixed buffer size

1. shared-memory bounded-buffer solution

```
struct item { ... };
#define BUFFER_SIZE 10
struct item buffer[BUFFER_SIZE];
int in = 0, out = 0;
```

(a). Producer

```
while (1) {
  while ((in + 1) % BUFFER_SIZE == out);
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
}
```

(b). Consumer

```
while (1) {
  while (in == out);
  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  return item;
}
```

Note that this code snippet CANNOT be used in inter-process environment, as there's no synchronizing mechanisms for these shared variables.

- buffer with size n will be full at $n - 1$.
- the same index in buffer might be concurrently read and written.
- `while` loop is time-consuming, we may use conditional variables, etc.

To solve the $n - 1$ issue:

```
struct item {};
#define BUFFER_SIZE 10
#define BUFFER_SIZE_MASKED (BUFFER_SIZE * 233)
/* the multiplier can be any number larger than 2 */

struct item buffer[BUFFER_SIZE];
int in = 0, out = 0;

struct item consumer() {
    while (in == out);
    int prev = out;
    out = (out + 1) % BUFFER_SIZE_MASKED;
    return buffer[prev % BUFFER_SIZE];
}

void producer(struct item i) {
    // for example, out = 1, in = 1 + 10 = 11, then the queue is full
```

```
            while (in == (out + BUFFER_SIZE) % BUFFER_SIZE_MASKED);
            buffer[in % BUFFER_SIZE] = i;
            in = (in + 1) % BUFFER_SIZE_MASKED;
        }
```

7.3 Ordinary Pipes

- communication between processes in consumer-producer scheme.

```
int mypipe[2];
// reader uses [0], writer uses [1]
if (pipe(mypipe)) {
  // error
  exit(1);
}
if (fork() == 0) {
  // child process as consumer, only use read-end
  close(mypipe[1]);
  char c;
  read(mypipe[0], &c, 1);
  // if next read returns 0, then there'll be no more data (EOF)
  assert(read(mypipe[0], &c, 1) == 0);
  // next time it will return -1
  assert(read(mypipe[0], &c, 1) == -1);
  close(mypipe[0]);
} else {
  // parent process as producer, only use write-end
  close(mypipe[0]);
  char c = '!';
  write(mypipe[1], &c, 1); // you may also use fprintf or fscanf
  close(mypipe[1]);        // EOF will be sent to reader
}
```

7.4 Indirect Communication

- mailbox or ports
  - each mailbox has a unique id

    - processes can communicate only if they share one mailbox
- primitives
    - `send(A, message)`
    - `receive(A, message)`

7.5 Socket

- an endpoint for communication
- in TCP/IP or Internet, socket is identified as `(ip addr, port)`
- communication takes place between a pair of / two sockets
- web server uses a fixed port (e.g. HTTP 80)
- client socket port is generally randomly generated
- ports < 1024 are used for pre-defined services (requires root or net privileges on Linux)

7.5.1 Create a Server-side Socket

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `read()` or `write()`
- `close()`
- study it yourself!

7.5.2 client

- `socket()`
- `connect()`

1. Q&A
   (a). How does process pick message from mailbox?
   Mailbox is a FIFO queue. It will pick the front.
   (b). What does fork do?
    - create new process
    - copy whole page table
    - write child pid into return value