CSE 30341
Operating System Principles

Lecture 4 – Processes

Recap – Last Lecture

- Operating System Services
- System Calls and Interrupts
- System Programs
- Operating System Design and Implementation
- System Layering
- Virtual Machines
Overview

• Process Concepts
• Process Scheduling
• Operations on Processes
• Inter-process Communication

Process Concept

• Job/Process/Task used interchangeably
• A process is an instance of a program ("program in execution")
• Program: “piece of code”
• Process: code + data + more data + control structures + ...
Process Memory Layout

- **Stack**: Temporary Data
  - Function parameters, local variables, function call frames.
- **Heap**: Dynamically allocated memory.
- **Data**: Global variables & constant strings.
- **Text**: Program code.

Process “Life Cycle”

- NEW
  - admitted
  - Interrupt
  - Scheduler dispatch
- READY
  - I/O or event completion
- RUNNING
  - I/O or event wait
- TERMINATED
  - exit
Process State

- A process changes **state** constantly:
  - **New**: being created.
  - **Running**: running on a processor core.
  - **Waiting**: waiting for an event.
  - **Ready**: (or runnable) waiting for a core.
  - **Terminated**: finished/dead.

OS Information about Processes

- Memory (Stack, Heap, Code, Static Data)
- Current State (e.g., program counter)
- Process ID
- Saved Registers
- Open Files
- Other bookkeeping data

This is called the **process control block (PCB)** or **task control block (TCB)**.
Process Control Block

```
struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;
#ifdef CONFIG_SMP
    struct list_node make_entry;
    int on_rq;
#endif
    int __on_rq;
    int prio, static_prio, normal_prio;
    unsigned int *priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_t_entity rt;
    #ifdef CONFIG_PREEMPT_NFIFIERS
    /* list of struct preempt_notifier */
    struct hlist_head preempt_notifiers;
    #endif
} //
```

PCB on Linux
CSE 30341 - Operating System Principles

```c
#include CONFIG_CC_STACKPROTECTOR

/* Canary value for the -fstack-protector gcc feature */
unsigned long stack_canary;

#endif

/* pointers to (original) parent process, youngest child, younger sibling,
   * older sibling, respectively.  (p->father can be replaced with
   * p->real_parent->pid)
   */
struct task_struct __rcu real_parent; /* real parent process */
struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() */

/* children/sibling forms the list of my natural children */
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list */

/* struct task_struct *group_leader; /* threadgroup leader */
/
/* pttrace is the list of tasks this task is using pttrace on.
   * This includes both natural children and PTRACE_ATTACH targets.

```

```c
/* CPU-specific state of this task */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* namespaces */
struct nsproxy *nsproxy;
/* signal handlers */
struct signal_struct *signal;
struct sigcand_struct *sighand;

/* sigset_t blocked, real_blocked;
   sigset_t saved_signal; /* restored if setrestore_save() was used */
/* struct sigpending pending;*/

unsigned long sas_sss_sp;
size_t sas_sss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
struct audit_context *audit_context;
#endif CONFIG_AUDITSCALL

uid_t loginuid;
unsigned int sessionid;
```

Process Switching ("Context Switch")

- Job Queue is all processes on the system.
- Ready Queue (Run Queue) contains processes waiting to run, or waiting for the CPU!
- Device Queue processes waiting for a device event ("blocked" devices).
- "Other" Queues contain processes waiting for other processes to finish, sleeping for time, etc.
Process “Wait” Queues

Schedulers

• **Long-term scheduler** which processes should be run in the future?
  – Degree of multiprogramming!

• **Short-term scheduler** which process should be run next?
  – Manages queues and quickly decides next process to run.
Scheduling Concerns

- Is enough RAM available to satisfy running processes?
- Is device throughput able to support more IO-bound processes?
- Is there enough CPU time available to satisfy all processes? (long) How do I schedule fairly? (short)
- Are there benefits for sleeping (swapping) a process for an extended time? (long)

Schedulers

- Short-term: invoked frequently (milliseconds); must be fast
- Long-term: infrequently (seconds)
- \textbf{I/O-bound process}: spends more time doing I/O than processing (CPU bursts can be frequent, but are short)
- \textbf{CPU-bound}: spends more time doing computations (very long CPU bursts)
Process Creation

- A process is always created via a parent, except for process `1, /sbin/init`.
- A parent can have multiple children. Entire structure is a tree.
- Each process has a unique identifier, the process identifier, or pid (get the pid of a process using the getpid() system call).
$p tty$
Process Creation

- **Resource sharing**
  1. Parent and children share all resources
  2. Children share subset of parent’s resources
  3. Parent and child share no resources

- **Execution**
  1. Parent and children execute concurrently
  2. Parent waits until children terminate

- **Address space**
  1. Child is duplicate of parent
  2. Child has a program loaded into it
UNIX Process Creation

- UNIX examples
  - `fork` system call creates new process
  - `exec` system call used after a `fork` to replace the process’ memory space with a new program

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    } else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```

Example Code
Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
  - Returns status data from child to parent (via **wait()**)
  - Process’ resources are deallocated by operating system

- Terminate a process from another process: **SIGKILL** (kill -9 pid)

- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

---

Process Termination

- What happens if a process “dies”?
  - The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the pid of the terminated process:
    \[
    \text{pid} = \text{wait}(&\text{status});
    \]
  - Parent “collects” child’s resources
  - If no parent waiting (did not invoke **wait()**) process is a **zombie**

- What happens if parent “dies”?
  - If parent terminated without invoking **wait**, process is an **orphan**
  - May receive “new parent” (grandparent, init process, etc.)
  - **Cascading termination**: no orphans allowed; if a process terminates, all its children must also be terminated
Interprocess Communication

- Processes **communicate** by sharing data.
- Why do processes communicate?
  - Computation speedup
  - Modularity
  - Information sharing
- Mechanism: **interprocess communication** (IPC)
- Two standard models: **Shared Memory** and **Message Passing**
Producer-Consumer Problem

- One process produces data. The second process consumes the data.
- Data stored in a buffer:
  - **Unbounded-Buffer** has no limit on size. Grows to size of memory.
  - **Bounded-Buffer** has fixed size. Creates a new problem:
    - *How do we handle the producer creating data too fast?*

---

Shared Memory Solution

**Circular buffer**

<table>
<thead>
<tr>
<th>NUL</th>
<th>NUL</th>
<th>NUL</th>
<th>NUL</th>
<th>NUL</th>
<th>NUL</th>
<th>NUL</th>
<th>NUL</th>
<th>NUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>OUT</td>
<td>IN</td>
<td>OUT</td>
<td>IN</td>
<td>OUT</td>
<td>IN</td>
<td>OUT</td>
<td>IN</td>
</tr>
</tbody>
</table>

IN = OUT -> EMPTY

A | NUL | NUL | NUL | NUL | NUL | NUL | NUL |

OUT | IN

NUL | NUL | C  | D  | E  | F  | NUL | NUL | NUL |

OUT | IN
Shared Memory Solution

- Shared data
  
  ```
  #define BUFFER_SIZE 10
  typedef struct {
      /*
      */
  } item;
  
  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  ```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded Buffer - Producer

```c
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```
Bounded Buffer - Consumer

```c
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

Message Passing

- Two primitives:
  - `send` (C, message) – messages have maximum size
  - `receive` (P, message)
- Think **mailboxes**.
- **Kernel** usually manages the message passing and “mailboxes”.
Message Passing Considerations

- How is the link established?
  - Automatically on send?
- Can the link be asymmetric?
  - Receiving a message: who is the sender?
- Is there a limit to the capacity of the link?
- Is the message size fixed or variable?
- Is a link unidirectional or bidirectional?
- Can there be multiple links between a pair of communication processes?

Message Passing

- Direct Communication
  - send (P, message) -> receiver process P
  - receive (Q, message) -> sender process Q
- Indirect Communication ("mailboxes")
  - send (M1, message) -> put in mailbox M1
  - receive (M1, message) -> take from mailbox M1
IPC Synchronization

• **Blocking?**
  – *Consumer* is put in a waiting scheduler queue if “mailbox” is empty.
  – *Producer* is put in a waiting scheduler queue if “mailbox” is full.

• **Non-blocking?**
  – Neither *Producer* nor *Consumer* blocks; failure is returned from message passing primitive instead.

Buffering

• Queue of messages attached to the link; implemented in one of three ways:
  – **Zero capacity** – no messages are queued on a link. Sender must wait for receiver (rendezvous)
  – **Bounded capacity** – finite length of $n$ messages. Sender must wait if link full
  – **Unbounded capacity** – infinite length. Sender never waits
**IPC - POSIX**

- POSIX Shared Memory
  
  ```c
  shm_id = shm_open(name, O_CREAT | O_RDWR, 0666);
  
  ftruncate(shm_id, 4096);
  
  shared_memory = (char *) shmat(shm_id, NULL, 0);
  
  sprintf(shared_memory, "Writing to shared memory");
  
  Also: shmdt (remove), shmctl (destroy)
  ```

**IPC - Mach**

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer
    ```c
    msg_send(), msg_receive(), msg_rpc()
    ```
  - Mailboxes needed for communication, created via `port_allocate()`
Recap

• Key Points:
  – System Layering
  – Concept of a Process
  – Scheduling
  – Process Creation and Termination
  – Interprocess Communication