Process

Joy Mukherjee

Indian Institute of Technology Bhubaneswar

What is a Process?

- A program by itself is not a process.
- A program is a passive entity (an executable file containing a list of instructions stored on disk).
- A **process** is an instance of a program in execution.
- A process is an active entity. The status of the current activity of a process is represented by the value of the program counter and the contents of the CPU registers.
- A program becomes a process when an executable file is loaded into memory.
- Multiple instances of the same program are different processes.
  - Text sections are equivalent.
  - Data, heap, and stack sections vary.

Memory Layout of a Process

Fixed Size	Command Line Arguments, and Environment Variables	High Address
Variable Size	Stack (Function Parameters, Return Addresses, and Local Variables)	
	↓ ↑	
Variable Size	Heap (Dynamically Allocated Memory)	
Fixed Size	Uninitialized Data (Global Variables)	
Fixed Size	Initialized Data (Global Variables)	
Fixed Size	Text (Executable Code)	Low Address

A C Program

```
#include <stdio.h>
#include <stdlib.h>
int x;                    // Uninitialized Data (Global Variables)
int y = 15;               // Initialized Data (Global Variables)
int main(int argc, char *argv[]) { // Command Line Arguments
    int *values;          // Stack (Local Variables)
    int i;
    values = (int *) malloc (sizeof(int) * 5); // Heap
    for(i = 0; i < 5; i++)
        values[i] = i;
    return 0;
}
```

Process control block (PCB)

- The primary data structure maintained by the kernel that contains information about a process.
- Kernel maintains a list of PCBs for all processes.
- Linux PCB (**task_struct**) has 150+ fields.

Contents of a PCB

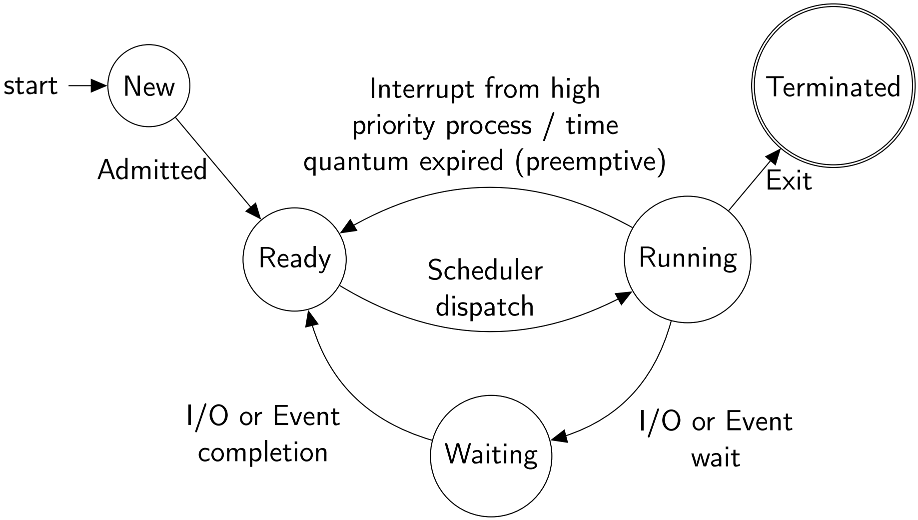
- **Process state**: New / Ready / Running / Waiting / Terminated.
- **Program counter**: The address of the next instruction to be executed for the process.
- **CPU registers**: Accumulators, index registers, stack pointers, and general-purpose registers, plus Program Status Word (PSW).
  - When an interrupt occurs, the program counter, the register contents must be saved to allow the process to be continued correctly afterward, when it is rescheduled to run.

- **Process-scheduling information**: Process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Process-synchronization information**: Sockets, Semaphores, Shared memory regions, Timers, Signal handlers.
- **Memory-management information**: The value of the base and limit registers and the page tables, or the segment tables.
- **Accounting information**: The amount of CPU and real time used, time limits, process id, parent process id, and so on.
- **I/O status information**: The list of I/O devices allocated to the process, a list of open files, and so on.
- Information for each thread.

States of a Process

- As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.
  - **New**
  - **Ready**
  - **Running**
  - **Waiting / Blocking**
  - **Terminated**
- Only one process can be running on any processor core at any instant.
- Many processes may be in ready and waiting states.

Process State Diagram



Process States

- **New**: The **long term scheduler** brings the process from secondary memory to the ready queue of RAM.
    - The **long term scheduler** controls the degree of multiprogramming (No. of processes in the RAM)
- **Ready**: The process is in the ready queue of RAM, and waiting to be assigned to a CPU based on some CPU scheduling algorithm (FCFS, SJF, SRTF, RR, Priority scheduling).
    - If the ready queue is full, the process is taken to the secondary memory by the **medium term scheduler**.
    - If the ready queue is not full, the process is brought to the ready queue by the **medium term scheduler**.

Process States

- **Running**: The **short term scheduler / dispatcher** selects the process from the ready queue of RAM and allocates the CPU for execution. In this state, the instructions of the process residing in RAM are being executed.
- **Waiting / Blocking**: The process is waiting for some event to occur such as completion of an I/O or reception of a signal in the waiting queue of RAM.
  - If waiting queue is filled up, the process is taken to the secondary memory by the **medium term scheduler**.
  - If the waiting queue has some space, the process is brought to the waiting queue by the **medium term scheduler**.
- **Terminated**: The process has finished execution. The **long term scheduler** brings the process into the secondary memory from the RAM.

Main Operations on a Process

- Process creation
- Process scheduling
- Process termination

Process Creation

- A process (parent) can create another process (child) via fork().
- Each process has a unique ID (an integer).
- PCB is created and initialized
- Initial resources allocated and initialized if needed
- Process added to ready queue (queue of processes ready to run).
- The new process can in turn create other processes, forming a tree of processes.

Process Creation

- The **systemd (earlier, init)** process (which always has a pid of 1) serves as the root parent process for all user processes.
- The **systemd** process is the first user process created when the system boots.
- Once the system has booted, the systemd process creates processes which provide additional services such as logind, udevd, and so on.
- **ps -el** lists all active processes in the system.
- **pstree** displays a tree of all processes in the system.

Process Creation

- **Resource sharing possibilities**:
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **Execution possibilities**:
  - Parent and children execute concurrently
  - Parent waits until children terminate
- **Memory address space possibilities**:
  - Address space of child is a duplicate of parent via fork(). Child inherits open files, privileges and scheduling attributes from the parent.
  - Child has a new program loaded into it via execlp(), execvp() etc.

Process Termination

- Process executes exit() / abort() and asks the operating system to terminate it.
- Process encounters a fatal error like arithmetic exception etc.
- Parent may terminate execution of children processes (ex. kill). Some possible reasons
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting, and the operating system may not allow a child to continue if its parent terminates.

Process Termination

- Parent may wait for the termination of a child by using the wait() system call.
- Parent obtains the exit status of the child in *status*.
- Returns the process identifier *pid* of the terminated child to the parent.

```
pid_t pid;
int status;
pid = wait(&status);
```

Process Termination

- When a process terminates,
  - All the resources of the process (physical and virtual memory, open files, and I/O buffers) are deallocated by the kernel.
  - However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.
  - A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process**.
  - All processes transition to zombie state when they terminate.
  - Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.

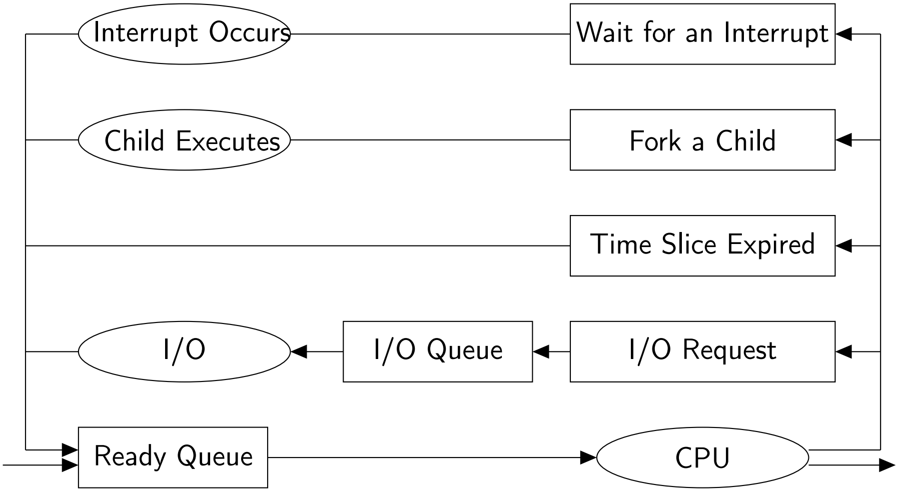
Process Termination

- If a parent did not invoke wait() and terminated,
    - Its child processes are orphans.
    - Linux assigns the **systemd process** (root of the process hierarchy in Linux system) as the new parent to orphan processes.
    - The systemd process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

Process Scheduling

- **Ready queue**: queue of PCBs of all processes residing in main memory that are ready to execute
- **Scheduler/Dispatcher**: picks up a process from ready queue according to some CPU Scheduling Policy and assigns it the CPU
- Selected process runs till
    - It needs to wait for some event to occur (ex. a disk read)
    - The CPU scheduling policy dictates that it be stopped
        - CPU time allotted to it expires (timesharing systems)
        - Arrival of a higher priority process
    - When it is ready to run again, it goes back to the ready queue
- Scheduler is invoked again to select the next process from the ready queue

Queueing Diagram Representation of Process Scheduling



Scheduling of CPU Scheduler

- The scheduler is scheduled by
  - the code associated with a hardware interrupt, or
  - code associated with a system call

Context of a Process

- Information that is required to be saved to be able to restart the process later from the same point
- Includes:
  - CPU state – all register contents, PSW
  - Program counter
  - Memory layout of a process
  - Open file information
  - Pending I/O and other event information
- In Linux, each process's context is described by a **task_struct** structure.

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Context-switch time is highly dependent on hardware support, e.g., the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers).
- Context-switch time is typically a few milliseconds.

Handling Interrupts

- The interrupt descriptor table (IDT) associates each interrupt identifier with an interrupt service routine (ISR).

Handling Interrupts

- The interrupt descriptor table (IDT) associates each interrupt identifier with an interrupt service routine (ISR).
- Jump to ISR

Handling Interrupts

- The interrupt descriptor table (IDT) associates each interrupt identifier with an interrupt service routine (ISR).
- Jump to ISR
- ISR first saves the context of the process

Handling Interrupts

- The interrupt descriptor table (IDT) associates each interrupt identifier with an interrupt service routine (ISR).
- Jump to ISR
- ISR first saves the context of the process
- Execute the ISR

Handling Interrupts

- The interrupt descriptor table (IDT) associates each interrupt identifier with an interrupt service routine (ISR).
- Jump to ISR
- ISR first saves the context of the process
- Execute the ISR
- Before leaving, ISR should restore the context of the process being executed

Handling Interrupts

- The interrupt descriptor table (IDT) associates each interrupt identifier with an interrupt service routine (ISR).
- Jump to ISR
- ISR first saves the context of the process
- Execute the ISR
- Before leaving, ISR should restore the context of the process being executed
- Return from ISR restores the PC

Handling Interrupts

- The interrupt descriptor table (IDT) associates each interrupt identifier with an interrupt service routine (ISR).
- Jump to ISR
- ISR first saves the context of the process
- Execute the ISR
- Before leaving, ISR should restore the context of the process being executed
- Return from ISR restores the PC
- ISR may invoke the dispatcher, which may load the context of a new process, which runs when the interrupt returns instead of the original process interrupted

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
- PCB of P_0 tells where to save

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
- PCB of P_0 tells where to save
- The dispatcher selects P_1 from ready queue

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
- PCB of P_0 tells where to save
- The dispatcher selects P_1 from ready queue
- The PCB of P_1 tells where the old state, if any, is saved

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
- PCB of P_0 tells where to save
- The dispatcher selects P_1 from ready queue
- The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
- PCB of P_0 tells where to save
- The dispatcher selects P_1 from ready queue
- The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1
- The dispatcher reloads the counter (timer) with T

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
- PCB of P_0 tells where to save
- The dispatcher selects P_1 from ready queue
- The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1
- The dispatcher reloads the counter (timer) with T
- The ISR returns, restarting P_1 (since P_1's PC is now loaded as part of the new context loaded)

Example: Time-sharing Systems

- Each process has a time quantum T allotted to it
- Dispatcher starts process P_0, loads an external counter (timer) that counts down from T to 0
- When the timer expires, the CPU is interrupted
- The ISR invokes the dispatcher
- The dispatcher saves the context of P_0
- PCB of P_0 tells where to save
- The dispatcher selects P_1 from ready queue
- The PCB of P_1 tells where the old state, if any, is saved
- The dispatcher loads the context of P_1
- The dispatcher reloads the counter (timer) with T
- The ISR returns, restarting P_1 (since P_1's PC is now loaded as part of the new context loaded)
- P_1 starts running