Process Management

A.Y. 2017-18

ACSO Tutoring

MSc Eng. Michele Zanella

Info

- Contacts:
  - michele.zanella@polimi.it
    - HEAP Lab – Campus Leonardo, via Golgi 39, Edificio 21, Piano 1, Ufficio 4, +39 02 2399 **9613**
      (send me an email to arrange for a meeting)
- Website:
  - https://beep.metid.polimi.it
- **Note for e-mail:**

  Subject: *[ACSO] your subject*

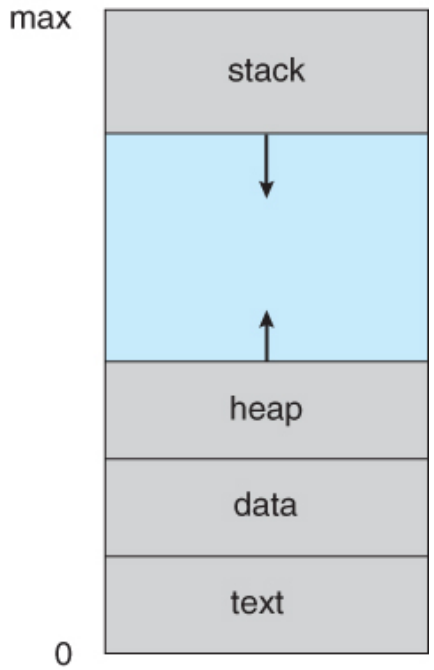
The process

A **process** is the unit of work in modern time-sharing system.

- System processes
- User processes

Process memory page:

- **Current status**:
  - Program counter
  - Processor registers
- **Text section**: Program code
- **Data section**: Global vars
- **Heap**: Dynamically allocated
- **Stack**: Temporary data



Process States

During execution a process can chenge its **state**:

- **New**: the process is being created
- **Running:** Instructions are being executed
- **Waiting:** the process is waiting for some event to occur (I/O, signal, …)
- **Ready:** The process is waiting to be assigned to a processor
- **Terminated**: The process has finished the execution

Only one process can be running on any processor at any instant.

Process Control Block

Information associated with a specific process are stored in the **Process Control Block (PCB)**:

- **Process state:** see previous slide
- **Program counter**: it indicates the address of the next istruction to be executed for the process
- **CPU Registers**: this information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward
- **CPU-Scheduling information**: e.g., process priority, pointers to scheduling queues and other parameters
- **Memory-management information**: e.g., page tables
- **Accounting information**: e.g., amount of CPU and real time used
- **I/O Status information**: e.g., list of I/O devices allocated to the process, a list of open files

Threads

- They allow processes to perform more than one task at a time
- Especially beneficial on multicore systems because of parallelism
- A basic unit of CPU utilization.
- PCB expanded to include information for each thread
- They shared with other threads belonging to the same process its code section, data section, and other OS resources.

Process Scheduling

Goals:

- **Multiprogramming**: having some process running at all times, to maximize CPU utilization
- **Time sharing**: switching the CPU among processes so frequently that users can interact with each program while it is running

Scheduler: it is in charge of selecting an available process for program execution on the CPU

Scheduling Queues

- **Job queue:** all processes in the system
- **Ready queue:** processes in main memory and ready that are waiting to be executed
- **Device queue:** processes waiting for a particolar I/O device

During the execution of a process one of the following events could occour:

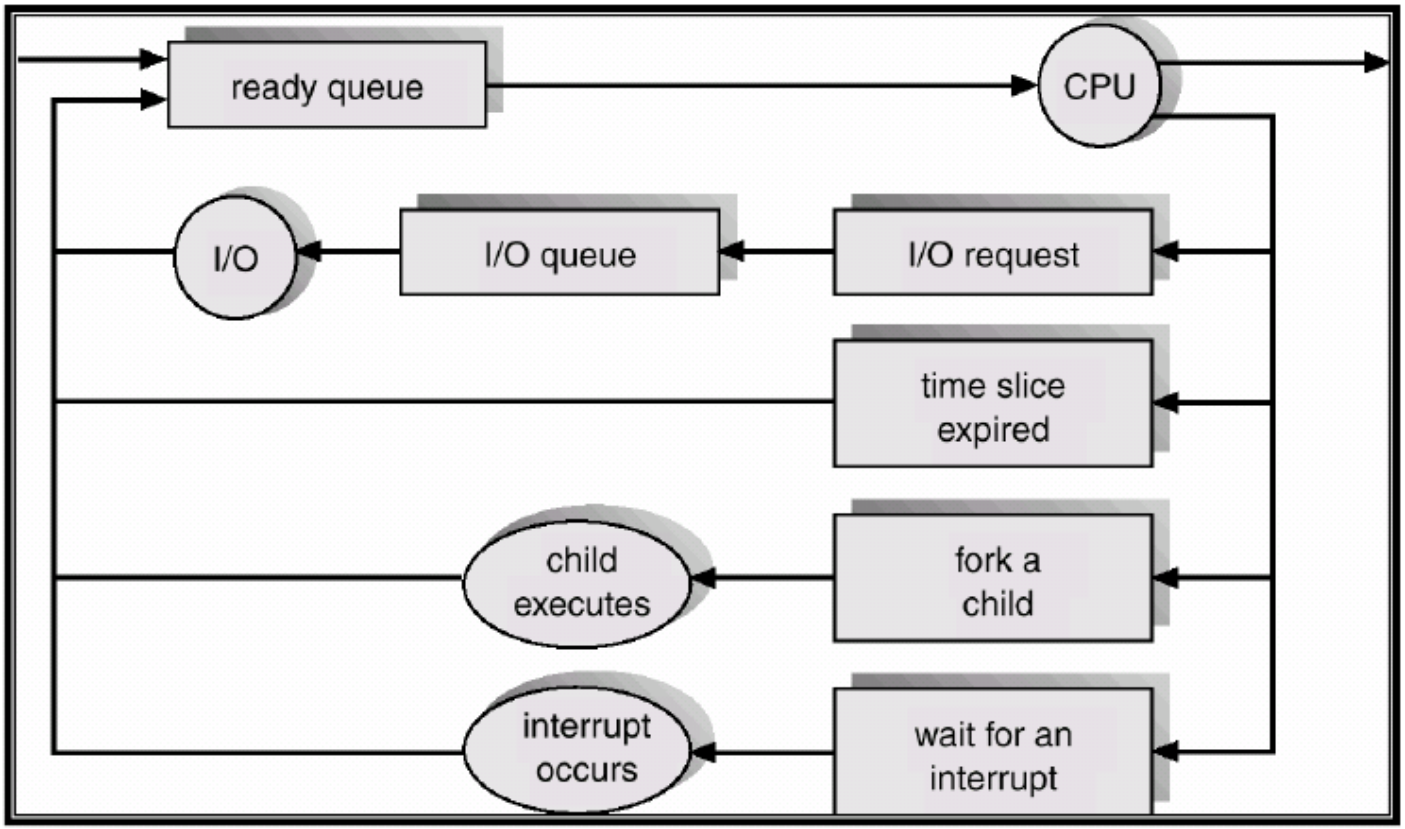
a) I/O request -> I/O queue
b) Creation of a new process and waiting for the child termination
c) Interrupt and removed forcibly from the CPU (e.g., time slice expiration)

Schedulers

It is in charge of selecting the processes from the queues to be executed. Two types:

- **Long-term (Job scheduler):** selects processes from the mass-storage device and loads them into memory for execution.
  It controls the *degree of multiprogramming*.
  It has to take into account if a process is **I/O-bound or CPU-bound** to select a good process mix.
- **Short-term (CPU scheduler):** selects from among the processes that are ready to execute and allocates the CPU to one of them.
  It must be fast.
- **Medium-term (Swapping scheduler):** removes a process from memory to reduce the degree of multiprogramming.

Schedulers


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

Context Switch

- Interrupts cause the OS to change a CPU from its current task and to run a kernel routine.
- The system has to save the current **context** of the running process to resume it correctly and to execute the new one-> **context switch**.

Operations on Processes

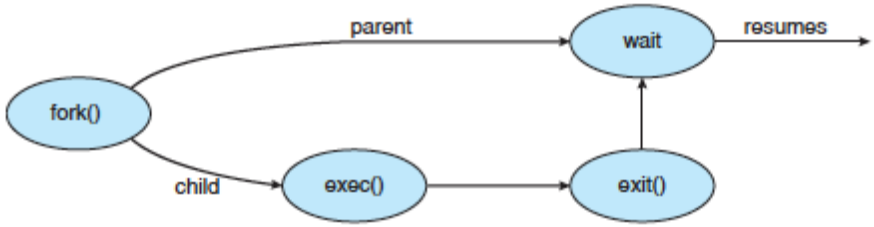
Process Creation

- The creating process: **parent process**
- **Process Identifier (PID):** used by the OS to identify uniquely a process.
- `init` process has PID 1
- During the process creation we have two possibilites:
    1. The parent continues the execution in parallele with the its children
    2. The parent waits until one of its children have terminated
- There is also two adress-space possibilities:
    1. The child process is a duplicate of the parent one
    2. The child process has a new program loaded into it

Operations on Processes

Process Creation

- `fork()`: it creates a new process that consists of a copy of the address space of the parent.
  It returns **0** for the child process and **the PID of the child** for the parent process.
- `exec()`: it replaces the process' memory space with a new program
- `wait()`: called by the parent to wait the termination of the child



Operations on Processes

Process Termination

- `exit()`: called by a process to ask the OS to delete it after it finishes executing its final statement.
  All the resources are deallocated.
- The parent can retrieve the exit status of a child by passing a parameter to the `wait` function
- **Zombie:** a process that has terminated, but whose parent has not yet called the `wait` function
- **Orphans:** a process whose parent did not invoke the `wait` and instead terminated.
  The child process is assigned to the `init` that will perform a `wait`

Multithreaded Programming

- Motivation
    1. Process creation is time consuming and resource intensive
    2. Threads play a vital role in a remote procedure call (RPC) system
    3. Most OS kernels re now multithreaded to perform specific task simultaneously.
- Benefits
    1. Responsiveness
    2. Resource sharing
    3. Economy
    4. Scalability

Multicore Programming

- Each core appears as a separate processor to the OS

- **Parallelism:** if a system can perform more than one task simultaneously
- **Concurrency:** when a system supports more than one task by allowing all the task to make progress by assigning a separate thread to each core

Programming challenges

- Identifying tasks
- Balance
- Data splitting
- Data dependency
- Testing and debugging

Type of parallelism

- **Data parallelism:** distributing a subsets of the same data across multiple computing cores and performing the same operation on each core
- **Task parallelism:** distributing tasks (threads) across multiple computing cores. Each thread is performing a unique operation

The Pthread Library

- API for creating and managing threads
- POSIX standard
- Any data declared globally are shared among all threads of the same process
- Separate threads begin execution in a specified function
- Include the `pthread.h` header file
- `pthread_t tid` declares the identifier for the thread
- `pthread_attr_t:` attributes for the thread set by the `pthread_attr_init(&attr)`
- `pthread_create(&tid,&attr,function,arg):` creates a separate thread passing the name of the function to begin the execution and its argument.

Synchronization

- **Cooperating system:** can effect or be affected by other processes executing in the system
- Concurrent access to shared data -> avoid data inconsistency
- Producer-Consumer problem -> the `counter` variable

```
while(true) {
    while(counter==100);
    buffer[in]=next_p;
    in=(in+1)%100;
    counter++;
}
```

```
while(true) {
    while(counter==0);
    next_c=buffer[out];
    out=(out+1)%100;
    counter--;
}
```

Synchronization

- **Problem!** Following the execution of these two statemets, the value of the variable `counter` may be 4,5 or 6!
- **Race Condition:** the outcome of the execution depends on the particular order in which the access take place
- We need to ensure that only one process at a time can be manipulating the variable `counter`

Critical Section

- **Critical Section**: segment of code in which the process may be changing common variables.
- When one process is executing in its critical section, no other process is allowed to execute in its critical section.
- **Entry section:** where the process require permission to enter its critical section
- **Exit section:** when the process exits its critical section
- A solution to the critical-section problem must satisfy three requirements:
    - **Mutual exclusion**
    - **Progress**
    - **Bounded waiting**
- **Preemptive kernels:** allows process to be preempted while it is running in kernel mode. More responsive and suitable for real-time programming
- **Non-preemptive kernels**: vice-versa

Peterson's Solution

- Requires the two processes to share two data items:
  - `turn`: indicates whose turin it is to enter the CS
  - `flag`: indicates if a process is ready to enter its CS
- To enter the CS, process P
  - sets `flag[i]=true`
  - sets `turn=j`

Mutex Locks (pthread)

- Protects CS: a process must acquire the lock before entering the CS and relase it when it exits the CS

- `pthread_mutex_t`: data type for mutex locks
- `pthread_mutex_init()`: creates a mutex
- `pthread_mutex_lock()`: mutex is acquired.
  If the mutex is unavailable, the calling thread is blocked until the owner releases it
- `pthread_mutext_unlock()`: mutex is released

Semaphore (POSIX)

- Is an integer variable that, a part for initialization, is accessed only through two standard atomic operations. Types:
  - **Counting**: used to control access to a given resource consisting of a finite number of instances
  - **Binary**: can range only between 0 and 1
  - **Named:** can be shared by multiple unrelated processes
  - **Unnamed**: can be used only by threads of the same process
- `sem_init()`: creates and initializes an unnamed semaphore. It is passed three parameters:
  - Pointer to the semaphore
  - A flag indicating the level of sharing
  - The semaphore's initial value
- `sem_wait()`: to acquire and decrement the semaphore
- `sem_post()`: to release and increment the semaphore

Deadlock and Starvation

- **Deadlock:** a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes**.**
  Necessary conditions:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Handling deadlocks:
  - Preventing or avoiding
  - Detecting deadlocks
  - Ignoring the problem assuming that deadlocks never occur
- **Starvation**: processes wait indefinitely within the semaphore

Exercise 1 (Thread & Parallelism)
21/11/2012

Condition	*local* in th_1	*local* in th_2
After stat. **A**	*EXISTS*	*EXISTS*
After stat. **C**	*CAN EXIST*	*EXISTS*
After stat. **D**	*DOESN'T EXIST*	*CAN EXIST*

Condition	*mess* {0,1}	*global* {0,1,2}
After stat. **A**	*1*	*0 / 2*
After stat. **B**	*0 / 1*	*1 / 2*
After stat. **C**	*0*	*1 / 2*
After stat. **D**	*0 / 1*	*1 / 2*

Exercise 1 (Thread & Parallelism)
21/11/2012

th_1	th_2	*global*
mutex_lock (law)	*sem_wait (mess)*	*2*