TDIU11

Operating Systems

Processes and Threads

[SGG7/8/9] Chapters 3.1-3.3 and 4.1-4.3

Christoph Kessler, IDA,
Linköpings universitet.

Processes and Threads – Overview

- **Process Concept**
  - Context Switch
  - Scheduling Queues
  - Creation and Termination
- **Cooperating Processes**
  - Interprocess Communication
  - Example: Bounded buffer in shared memory
- **Thread Concept**
- **Multithreading Models**
- **Threading Issues**

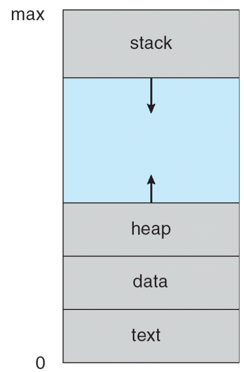
Process Concept

- **Process** = a program in execution
  - Program is a ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Example: Consider multiple users executing the same program
- Textbook uses the terms *job* and *process* almost interchangeably.

Process Concept

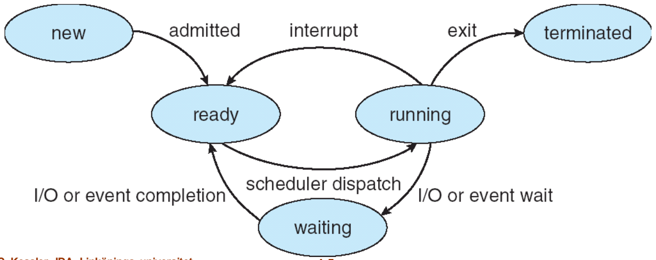
- **Process** = a program in execution
- Needs resources for execution
  - esp., CPU, memory slice
- A **process** includes:
  - The program code (also called **text section**)
  - Current activity including **program counter**, processor **registers**
  - **Data section** containing global variables
  - **Stack** containing temporary data: function parameters, return addresses, local variables
  - **Heap** containing memory dynamically allocated during run time

A process in memory:



Process State

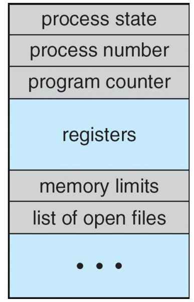
- As a process executes, it changes ***state***
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a process
  - **terminated**: The process has finished execution



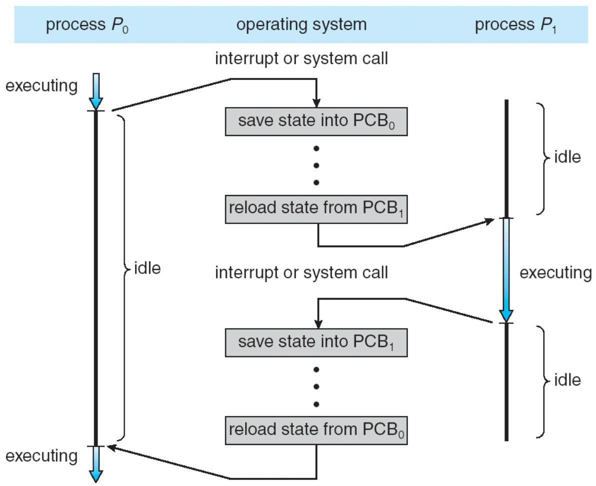
Process Control Block (PCB)

A data structure for each process in the OS kernel, containing information associated with a process (**PCB**, also called **task control block**)

- Process **state** – running, waiting, etc.
- Program **counter** – location of instruction to execute next
- CPU **register contents** of all process-centric CPU registers
- CPU **scheduling information** d – priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



CPU Switch From Process to Process

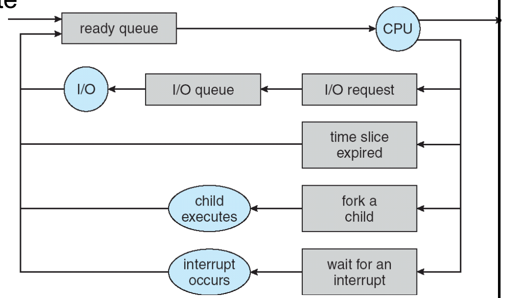


Context Switch

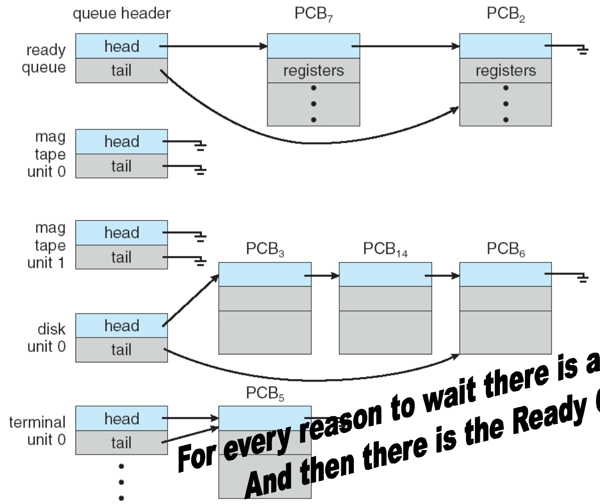
- When CPU switches to another process, the system must
  - save the state of the old process
  - and load the saved state for the new process
- Context-switch time is **overhead**
  - the system does no useful work while switching
  - time depends on hardware support

Process Scheduling Queues

- **Job queue**
  - set of all processes in the system
- **Ready queue**
  - set of all processes residing in main memory, ready and waiting to execute
- **Device queues**
  - set of processes waiting for an I/O device
- Processes migrate among the various queues



Ready Queue And Various I/O Device Queues



For every reason to wait there is also a queue. And then there is the Ready Queue.

Schedulers

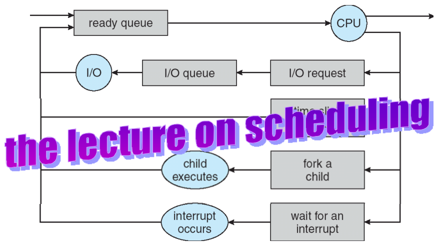
- **Long-term scheduler** (or **job scheduler**)
  - for batch systems – new jobs for execution queued on disk
  - selects which processes should be brought into the ready queue, and loads them into memory for execution
  - controls the *degree of multiprogramming*
  - invoked very infrequently (seconds, minutes)
  - No long-term scheduler on UNIX and Windows; instead **swapping**, controlled by **medium-term scheduler**
- **Short-term scheduler** (or **CPU scheduler**)
  - selects which ready process should be executed next
  - invoked very frequently (milliseconds) $\Rightarrow$ must be fast

CPU-bound vs I/O-bound processes

- **I/O-bound process**
  - spends more time doing I/O than computations
  - many short CPU bursts
- **CPU-bound process**
  - spends more time doing computations;
  - few very long CPU bursts
- Long-term (or medium-term) scheduler should aim at a good **process mix.**

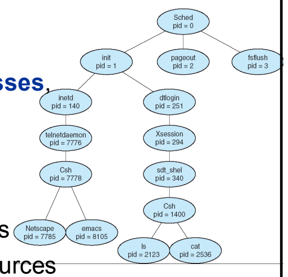
Scheduling

- **Non-preemptive scheduling:**
  - process keeps CPU until it terminates or voluntarily releases it (sleep() – step back into ready queue)
- **Preemptive scheduling:**
  - OS puts process from CPU back into ready queue after a certain time quantum has passed



More about this in the lecture on scheduling

Process Creation



- **Parent process** creates **children processes**, which, in turn create other processes, forming a **tree of processes**
- **Resource sharing** variants:
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **Execution** variants:
  - Parent and children execute concurrently
  - Parent waits until children terminate
- **Address space** variants:
  - Child is a duplicate of parent
  - Child has a program loaded into it

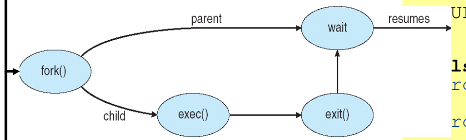
Example: Process Creation in UNIX

- **fork** system call
  - creates new child process
- **exec** system call
  - used after a **fork** to replace the process' memory space with a new program
- **wait** system call
  - by parent, suspends parent execution until child process has terminated

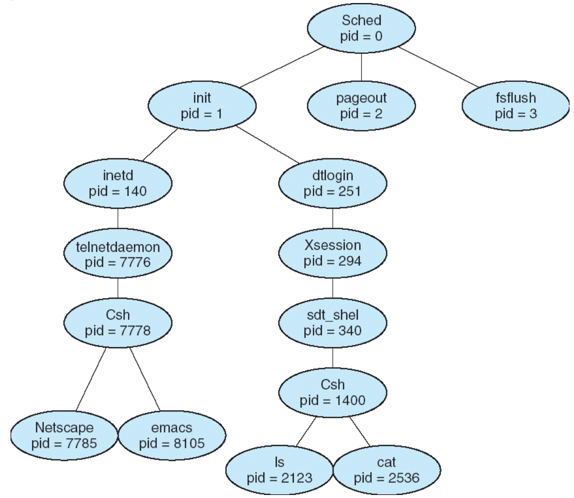
C program forking a separate process

```
int main()
{
    Pid_t  ret;
  /* fork another process: */
  ret = fork();
  if (ret < 0)  {  /* error
  occurred */
      fprintf ( stderr, "Fork
  Failed" );
      exit(-1);
  }
  else if (ret == 0)  { // I am
  child process:
      execlp ( "/bin/ls", "ls",
   ULL );

   lse  {  // I am the parent
   rocess
                // of child
   rocess with PID==ret
      /* wait for child to
```



A typical tree of processes in Solaris



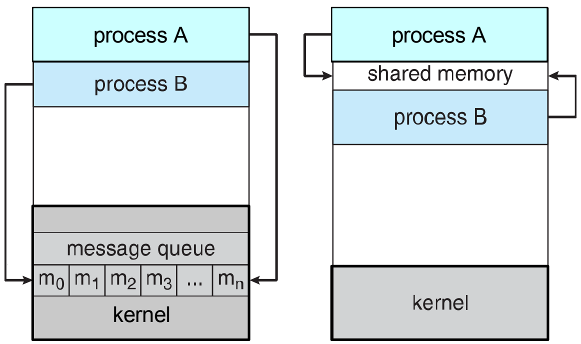
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Process returns status value to its parent (used in **wait**)
  - OS de-allocates process's resources
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
- If parent is exiting:
  - Some OS do not allow child to continue after parent terminates
    - All children terminated - *cascading termination*

Cooperating Processes

- **Independent** process
  - cannot affect or be affected by execution of another process
- **Cooperating** process
  - can affect or be affected by execution of another process
- **Advantages of process cooperation**:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- **Inter-Process Communication (IPC)**
  - shared memory
  - message passing
  - signals

IPC Models – Realization by OS



(a) IPC via **Message Passing**

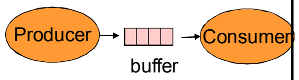
(b) IPC via **Shared Memory**

Example: POSIX Shared Memory API

- #include <sys/shm.h>
  #include <sys/stat.h>
- Let OS create a shared memory segment (system call):
  - int segment_id = **shmget** (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
- Attach the segment to the executing process (system call):
  - void *shmemptr = **shmat** (segment_id, NULL, 0);
- Now access it:
  - strcpy ((char *)shmemptr, "Hello world"); // Example: copy a string into it
  - …
- Detach it from executing process when no longer accessed:
  - **shmdt** (shmemptr);
- Let OS delete it when no longer used:
  - **shmctl** (segment_id, IPC_RMID, NULL);

Example for IPC: Producer-Consumer Problem

- **Producer-Consumer paradigm** for cooperating processes:
  - *producer* process produces data items that are consumed by a *consumer* process
- **Realization with shared memory:** Shared buffer (queue) of data items



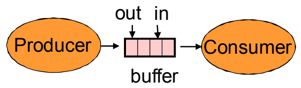
  - *unbounded-buffer*
    - places no practical limit on the size of the buffer
    - Consumer must wait when buffer is empty
  - *bounded-buffer*
    - assumes that there is a fixed buffer size
    - Producer must also wait when buffer is full

Bounded-Buffer – Shared-Memory Solution

- Shared buffer:

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```



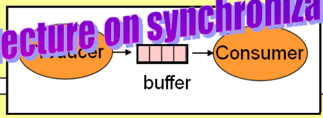
- buffer empty when in == out
- buffer full when ((in+1) % BUFFER_SIZE) == out
- can hold at most BUFFER_SIZE – 1 elements

Bounded-Buffer Producer and Consumer

Producer code

```
while (true)  {
     /* ... produce an item */
        while (((in + 1) % BUFFER SIZE)   == out)
              ;   /* do nothing -- no free buffers
  */
     buffer[in] = item;
     in = (in + 1)
```

To be continued in TDIU16 lecture on synchronization.

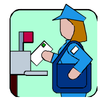



```
while (true) {
        while (in == out)
              ; // do nothing -- nothing to
  consume
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
     //  ... now use the item;
}
```

Consumer code

IPC with Message Passing

- **Message system**
  - processes communicate with each other without resorting to shared variables
  - provides two basic operations:
    - **send**(*receiverPID*, *message*)
    - **receive**(*senderPID*, *message*)
- In order to communicate, two processes
  - establish a *communication link* between them
  - exchange messages via send/receive



- [SGG7] 3.4.2.
- More about message passing variants and programming in TDDC78 *Programming of Parallel Computers*

Client-Server Communication

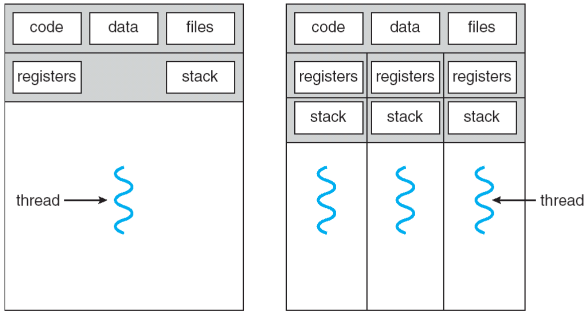
- **Message passing variant for client-server systems**
- **Sockets**
  - Endpoint for IPC between clients and servers
  - addressed by (*IP address*, *port number*) instead of PID
- **Remote Procedure Calls**
  - Client calls function of (maybe remote) server process by sending a RPC request to a server socket address
  - Server listens on socket port for incoming RPC requests
- In Java: *Remote Method Invocation (RMI)*

- [SGG7] 3.6

Threads – Overview

- Thread Concept
- Multithreading Models
- Threading Issues
- Thread libraries
  - Pthreads [SGG7] 4.3.1
  - Win32 Threads [SGG7] 4.3.2
  - Java Threads [SGG7] 4.3.3
- OS thread implementations
  - Windows XP Threads [SGG7] 4.5.1
  - Linux Threads [SGG7] 4.5.2

Single- and Multithreaded Processes



single-threaded process

multithreaded process

A **thread** is a basic unit of CPU utilization:

- Thread ID, program counter, register set, stack.
- May be represented in a Thread Control Block (TCB)

A process may have one or several threads.

Threads: Motivation

- Most modern applications are **multithreaded**
  - Several threads can run within an application, and thus, within a process
- Multiple **tasks** with the application can be implemented by separate threads
  - Example: Tasks in a web browser
    - Update display
    - Fetch data
    - Spell checking
    - Answer a network request
- Can simplify code, increase efficiency / responsiveness

Benefits of Multithreading

- Responsiveness
  - Interactive application can continue even when part of it is blocked
- Resource Sharing
  - Threads of a process share its memory by default.
- Economy
  - Light-weight
  - Creation, management, context switching for threads is much faster than for processes
    - E.g. Solaris: creation 30x, switching 5x faster
- Utilization of Multiprocessor Architectures
  - Threads are more convenient for shared-memory parallel processing on multiprocessors, such as multi-core CPUs, to speed-up program execution

User Threads (User-Level Threads)

- Thread management (scheduling, dispatch) done by user-level threads library (linked with the application), **without kernel support.**
- The thread-unaware kernel views all user threads of a multithreaded process as a single thread of control.
  - process dispatched as a unit

☺ user control of scheduling algorithm; less overhead

☹ user threads do not scale well to multiprocessor systems

- Three primary user-level thread libraries:
  - Win32 threads
  - Java threads
  - POSIX Pthreads (API / standard, not implementation – may be provided as either user- or kernel-level library)

Kernel Threads (Kernel-Level Threads)

- **Threads are managed by the OS kernel** (Kernel-specific thread API)
- Each kernel thread services (executes) one or several user threads

☺ Flexible: OS can dispatch ready threads of a multithreaded process even if some other thread is blocked.

☹ Kernel invocation overhead at scheduling/synchronization; less portable

- All modern operating systems support kernel-level threads

In short:
Kernel threads = **kernel-managed** threads.

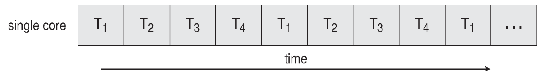
NB – The term "kernel thread" is sometimes misused with a different meaning, namely for the part of a program thread doing a syscall and thus running in kernel mode. This is *wrong* usage of the term and has nothing to do with the above kernel-thread/user thread concept!

Multicore Programming with Threads

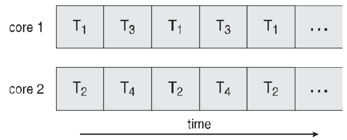
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include: **Dividing activities, Balance, Data splitting, Data dependency, Testing and debugging**
- ***Parallelism*** implies that a system can perform more than one task simultaneously, using multiple processors
  - Program designed with multiple processors in mind
- ***Concurrency*** supports more than one task making progress
  - Also on single processor / core, scheduler providing concurrency
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- More about this in course TDDD56 Multicore and GPU Programming

Concurrency vs. Parallelism

Concurrent execution on single-core system:



Parallelism on a multi-core system:



Side remark: Amdahl's Law

- Estimates performance gains from adding additional cores to an application that does both serial and parallel(izable) work
- *S* is serial portion of the work
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel(izable) / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Multithreading Models

Relationship user threads – kernel threads:

- **Many-to-One (M:1)**
- **One-to-One (1:1)**
- **Many-to-Many (M:N)**
- Variations:
  - Two-Level Model
  - Light-Weight Processes [SGG7] 4.4.6

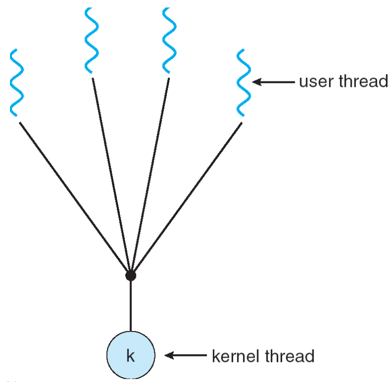
Many-to-One

- Many user-level threads mapped to single kernel thread

☺ Low overhead

☹ Not scalable to multiprocessors

- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
- Few current OS support this model



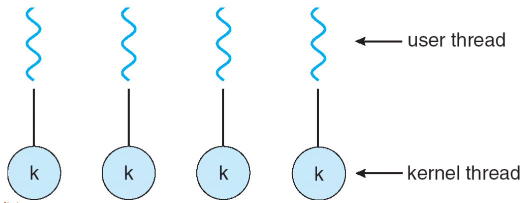
One-to-One

- Each user-level thread maps to one kernel thread

☺ more concurrency; scalable to multiprocessors

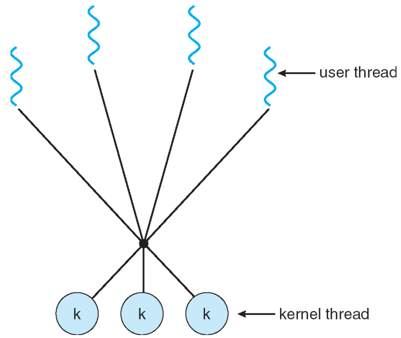
☹ overhead of creating a kernel thread for each user thread
  (can partly be eliminated by using *thread pools*)

- The preferred model for parallel computing on multicore CPUs
  - Many modern OS support it



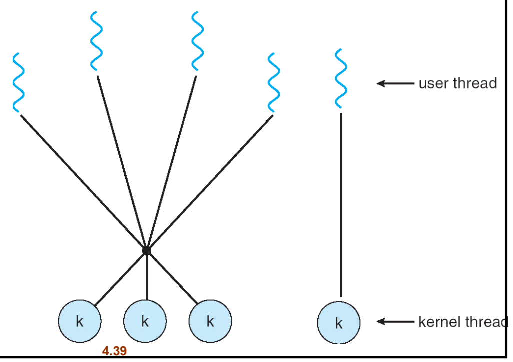
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Solaris 8 and earlier



Two-level Model

- Similar to Many-to-Many, except that it allows a user thread to be **bound** to a kernel thread
- Examples
  - Solaris 8 and earlier
  - IRIX
  - HP-UX
  - Tru64 UNIX



What have we learned?

- Processes versus Threads
- Process control block
- Context switch
- Ready queue and other queues used for scheduling
- Long-/Mid-term versus Short-term scheduler
- Process creation and termination
- Process tree
- Inter-Process Communication
- Motivation for multithreading a process
- Thread control block
- User (level) threads versus Kernel (level) threads
- Threading models: M:1, 1:1, M:N, two-level